What is C in practice? (Cerberus survey v2): Analysis of Responses - with Comments (n2015)

Kayvan Memarian and Peter Sewell

University of Cambridge

2016-03-08

Total responses: 323
  main2.2  : 20
  main2    : 179
  freebsd  : 9
  gcc      : 5
  google-comp : 1
  google   : 50
  libc     : 8
  linux    : 0
  llvm     : 8
  msvc     : 0
  regehr-blog : 34
  x11      : 7
  xen      : 4

Where did you come across this survey?
  gcc mailing list          :  11  (15%)
  llvm mailing list         :  49  (69%)
  Linux kernel mailing list :   1  ( 1%)
  FreeBSD mailing list      :   9  (12%)
  netos group               :   1  ( 1%)
  no response               :  84
  other                     : 176
  - internal list
  - Blog
  - blogpost about the finished survey, but I didn’t read the results yet!!
  - colleague
  - A friend shared it with me
  - coworker
  - ARM mailing list
  - Facebook
  - twitter
  - Facebook
  - Friend
  - forwarded by a colleague
  - Facebook
  - NetBSD developer referral
  - EuroLLVM
  - facebook
  - facebook share
  - colleague
  - forwarded by a friend
  - work internal mailing list
  - mailing list at work
  - internal mailing list
  - internal company mailing list
  - google mailing list
  - google internal list
  - internal google mailing list
  - internal Google mailing list
  - internal mailing list at work
  - EuroLLVM
  - llvm weekly
  - llvm mailing list, EuroLLVM
  - Google employee internal list
- FreeNode IRC
- ##c
- LLVM Weekly issue 68
- llvm mailing list, llvmweekly
- Company mailing list
- Internet forum
- Internal company mailing list
- Google internal list
- internal mailing list
- llvm blog
- llvm mailing list, EuroLLVM
- twitter
- Tony Finch via twitter: https://twitter.com/fanf/status/590464941527801856
- IRC channel
- LLVM weekly
- IRC
- LLVM Weekly repost on the LLVM blog
- LLVM weekly
- #chiark
- LLVM Weekly
- IRC channel
- Through a colleague
- Twitter (@fanf)
- reddit
- reddit.com
- llvm weekly blog RSS feed
- friend
- reddit
- friend
- company chatroom
- probably 4chan
- reddit
- llvm weekly
- Reddit
- llvm mailing list, reddit
- friend
- Colleague referral.
- reddit
- llvm website
- blog.llvm.org
- twitter
- Reddit
- "This Week in LLVM" http://llvmweekly.org/issue/67 (via lwn.net)
- Planet Clang
- Forwarded to our team by our manager, who probably got it from an LLVM mailing list.
- Google+
- LLVM weekly summary
- EuroLLVM
- blog post
- EuroLLVM
- Someone at work forwarded it to me
- reposted by someone
- google internal mailing list
- google group mail
- "eng-misc@google" mailing list
- google internal
- eng-misc
- xen-devel
- Xen development mailing list
- xen-devel

==================================

PEOPLE SUMMARY

==================================

===

main2.2:0       2015/06/15
main2.2:1       2015/06/16
main2.2:2       2015/06/30
main2.2:3       2015/06/30
main2.2:4       2015/06/30
main2.2:5       2015/06/30
main2.2:6       2015/06/30
main2.2:7       2015/06/30
main2.2:8       2015/07/08
main2.2:9       2015/07/09
main2.2:10      2015/07/13
main2.2:11      2015/07/14
main2.2:12      2015/07/15
main2.2:13      2015/07/19
main2.2:14      2015/07/27
main2.2:15      2015/07/31
main2.2:16      2015/08/01
main2.2:17      2015/08/19
main2.2:18      2015/08/28
main2.2:19      2015/09/29
main2:0         2015/04/10
main2:1         2015/04/13
main2:2         2015/04/13
main2:3         2015/04/13
main2:4         2015/04/13
main2:5         2015/04/13
main2:6         2015/04/13
main2:7         2015/04/14
main2:8         2015/04/14
main2:9         2015/04/14
main2:10        2015/04/14
main2:11        2015/04/14
main2:12        2015/04/14
main2:13        2015/04/14
main2:14        2015/04/14
main2:15        2015/04/15
main2:16        2015/04/15
main2:17        2015/04/15
main2:18        2015/04/15
main2:19        2015/04/15
main2:20        2015/04/15
main2:21        2015/04/15
main2:22        2015/04/15
main2:23        2015/04/15
main2:24        2015/04/15
main2:25        2015/04/15
main2:26        2015/04/15
main2:27        2015/04/15
main2:28        2015/04/15
main2:29        2015/04/15
main2:30        2015/04/15
main2:31        2015/04/15
main2:32        2015/04/15
main2:33        2015/04/15
main2:34        2015/04/16
main2:35        2015/04/16
main2:36        2015/04/16
main2:37        2015/04/16
main2:38        2015/04/16
main2:39        2015/04/16
main2:172  2015/05/07
main2:173  2015/05/15
main2:174  2015/05/15
main2:175  2015/05/22
main2:176  2015/05/22
main2:177  2015/05/31
freebsd:0  2015/04/25
freebsd:1  2015/04/25
freebsd:2  2015/04/25
freebsd:3  2015/04/25
freebsd:4  2015/04/26
freebsd:5  2015/04/26
freebsd:6  2015/05/14
freebsd:7  2015/04/26
freebsd:8  2015/04/28
gcc:0     2015/04/17
gcc:1     2015/04/17
gcc:2     2015/04/18
gcc:3     2015/04/21
gcc:4     2015/04/23
google-comp:0  2015/05/26
google:0   2015/05/26
google:1   2015/05/26
google:2   2015/05/26
google:3   2015/05/26
google:4   2015/05/26
google:5   2015/05/26
google:6   2015/05/26
google:7   2015/05/26
google:8   2015/05/26
google:9   2015/05/26
google:10  2015/05/26
google:11  2015/05/26
google:12  2015/05/26
google:13  2015/05/26
google:14  2015/05/26
google:15  2015/05/26
google:16  2015/05/26
google:17  2015/05/26
google:18  2015/05/26
google:19  2015/05/26
google:20  2015/05/26
google:21  2015/05/26
google:22  2015/05/26
google:23  2015/05/26
google:24  2015/05/26
google:25  2015/05/26
google:26  2015/05/26
google:27  2015/05/26
google:28  2015/05/26
google:29  2015/05/26
google:30  2015/05/26
google:31  2015/05/26
google:32  2015/05/26
google:33  2015/05/26
google:34  2015/05/26
google:35  2015/05/27
google:36  2015/05/27
google:37  2015/05/27
google:38  2015/05/27
google:39  2015/05/27
google:40  2015/05/27
google:41  2015/05/27
google:42  2015/05/27
google:43  2015/05/29
google:44  2015/06/01
google:45  2015/06/01
google:46  2015/06/01
google:47  2015/06/03
google:48  2015/06/04
libc:0    2015/05/26
libc:1    2015/05/26
libc:2    2015/05/26
libc:3    2015/05/27
libc:4    2015/05/27
libc:5    2015/05/27
libc:6    2015/05/27
libc:7    2015/06/05
llvm:0    2015/04/23
llvm:1    2015/04/23
llvm:2    2015/04/23
llvm:3    2015/04/24
llvm:4    2015/04/24
llvm:5    2015/05/04
llvm:6    2015/05/05
llvm:7    2015/07/02
regehr−blog:0 2015/04/21
regehr−blog:1 2015/04/21
regehr−blog:2 2015/04/21
regehr−blog:3 2015/04/21
regehr−blog:4 2015/04/21
regehr−blog:5 2015/04/21
regehr−blog:6 2015/04/22
regehr−blog:7 2015/04/22
regehr−blog:8 2015/04/22
regehr−blog:9 2015/04/22
regehr−blog:10 2015/04/22
regehr−blog:11 2015/04/23
regehr−blog:12 2015/04/23
regehr−blog:13 2015/04/23
regehr−blog:14 2015/04/23
regehr−blog:15 2015/04/23
regehr−blog:16 2015/04/23
regehr−blog:17 2015/04/23
regehr−blog:18 2015/04/24
regehr−blog:19 2015/04/24
regehr−blog:20 2015/04/24
regehr−blog:21 2015/04/25
regehr−blog:22 2015/05/01
regehr−blog:23 2015/05/02
regehr−blog:24 2015/05/05
regehr−blog:25 2015/05/26
regehr−blog:26 2015/05/26
regehr−blog:27 2015/06/02
regehr−blog:28 2015/06/03
regehr−blog:29 2015/06/26
regehr−blog:30 2015/08/27
regehr−blog:31 2015/09/07
regehr−blog:32 2015/09/07
regehr−blog:33 2015/09/07
x11:0     2015/05/26
x11:1     2015/05/26
x11:2     2015/05/26
x11:3     2015/05/27
x11:4     2015/05/27
x11:5     2015/05/27
x11:6     2015/05/30
xen:0     2015/05/26
xen:1     2015/05/27
xen:2     2015/05/27
xen:3     2015/06/03

========================================================
Your expertise [click all that apply]:

- C applications programming: 255 (22%)
- C systems programming: 230 (19%)
- Linux developer: 160 (13%)
- Other OS developer: 111 (9%)
- C embedded systems programming: 135 (11%)
- C standard: 70 (6%)
- C or C++ standards committee member: 8 (0%)
- Compiler internals: 64 (5%)
- GCC developer: 15 (1%)
- Clang developer: 26 (2%)
- Other C compiler developer: 22 (1%)
- Program analysis tools: 44 (3%)
- Formal semantics: 18 (1%)
- no response: 6
- other: 18

- Compiler internals, Clang developer, C++ applications developer
- C applications programming, C systems programming, Other OS developer, C variables with non-integer pointers
- C systems programming, Other OS developer, C embedded systems programming, Program analysis tools, Compiler tester
- just a humble C++ developer who doesn't mess around with these details
- C applications programming, C systems programming, Linux developer, C standard, Compiler internals, Generic compiler development
- C++ app / systems programming
- C applications programming, C systems programming, Linux developer, Other OS developer, C embedded systems programming, Compiler internals, Compilers which target C as an intermediate language
- i literally just fuck around for a bit with gcc and strace, and go back to python
- C systems programming, Linux developer, C embedded systems programming, C standard, Compiler internals, C-code generation from other languages
- C applications programming, C systems programming, Linux developer, Other OS developer, Game Console Developer
- C applications programming, Compiler internals, Parallel programming, GPU programming
- C applications programming, C systems programming, Other OS developer, C embedded systems programming, Program analysis tools, Crypto
- C applications programming, C systems programming, Linux developer, C standard, C standard library implementation
- C systems programming, C standard, Linux userspace, libc
- C applications programming, C systems programming, Linux developer, Other OS developer, C standard, Compiler internals, Other C compiler developer, Program analysis tools, Formal semantics, sometime security coordinator

====================================================================

=== PEOPLE DETAILS ===

main2.2:0 2015/06/15
- compilers: MSVC
- systems contributed to:

main2.2:1 2015/06/16
- compilers: 
- systems contributed to:

main2.2:2 2015/06/30
- compilers: GCC
- avr-gcc
- systems contributed to:

main2.2:3 2015/06/30
- compilers: gcc
- Windriver
- systems contributed to: Automotive embedded systems

main2.2:4 2015/06/30
compilers: GCC, Clang, Frama-C
systems contributed to: GCC

main2.2:5 2015/06/30
compilers: gcc, g++, clang, msvc
systems contributed to:

main2.2:6 2015/06/30
compilers: gcc
g++
clang
clang++
c (MSVC)
systems contributed to:  Linux,
Mac OS X,
Windows

main2.2:7 2015/06/30
compilers: GCC, Clang/LLVM
systems contributed to:

main2.2:8 2015/07/08
compilers: GCC (since 1988), clang, MSVC, Cadence Xtensa XCC, Turbo C, some embedded cpu compilers (such as Keil, etc).
systems contributed to:

main2.2:9 2015/07/09
compilers: clang
gcc
msvc
systems contributed to:

main2.2:10 2015/07/13
compilers: GCC
-O0 for debug builds, in order to minimize deviation of the actual behavior of the emitted machine code from that of the source (i.e. disallow transformations based on the "as-if rule").
-O2 or -O3 for release builds on desktop and desktop-class (e.g. modern mobile OS) systems in order to optimize my executables for execution speed
-Os for release builds of embedded systems projects (e.g. microcontrollers @M-^@ M-^S I use AVR-gcc quite often). This is sometimes needed to have the executable fit into the code memory of the device.
-fstrict-aliasing: I turn on this flag because in my code I do not rely on constructs that violate strict aliasing, and some day I expect the compiler to generate warnings when I accidentally do and this flag is turned on.
-flto: for release builds, I heavily rely on whole-program optimization so that I can avoid explicitly inlining functions and falling back to ugly macros.

Clang: basically the same as above, excepting that I do not use Clang for near-metal development.

systems contributed to:  I program mainly in C and Objective-C (the latter for iOS). My biggest current C programming project is an interpreter for a scripting language I develop: http://github.com/H2CO3/Sparkling.

For iOS, I used to make tweaks (supplementary dynamic components/"plug-ins" for jailbroken devices); these usually did not require reasoning about edge cases of the language, as Objective-C permits a higher lever of abstraction than plain C; less hacking is required to implement individual features.

I have also begun to implement a compiler for a C-like language, and for that, I’ve extensively studied some of the most practical problems in the area of compiler implementation; I have not yet, however, made any significant contributions to existing C compilers.
main2.2:11  2015/07/14
  compilers: gcc
  systems contributed to:
main2.2:12  2015/07/15
  compilers: gcc
  systems contributed to:
main2.2:13  2015/07/19
  compilers: gcc
  clang
  systems contributed to: GNU/Linux
main2.2:14  2015/07/27
  compilers: gcc
  systems contributed to:
main2.2:15  2015/07/31
  compilers: gcc
  clang
  -fundefined-trap
  systems contributed to:  iOS apps
main2.2:16  2015/08/01
  compilers: gcc, gdb, dwarves, elfutils
  systems contributed to:  linux
main2.2:17  2015/08/19
  compilers: gcc -Og -g -fno-strict-aliasing
  clang (sometimes)
  systems contributed to:
main2.2:18  2015/08/28
  compilers: gcc, xlc
  systems contributed to:
main2.2:19  2015/09/29
  compilers: gcc
  systems contributed to:
main2:0     2015/04/10
  compilers: gcc
  systems contributed to:
main2:1     2015/04/13
  compilers: GCC, Clang, IAR C/C++, MSVC
  systems contributed to:
main2:2     2015/04/13
  compilers: gcc, clang, sparse
  -fno-strict-aliasing and -fno-delete-null-pointer-checks for legacy low-level code
  systems contributed to:
main2:3     2015/04/13
  compilers: gcc, clang, visual studio
  systems contributed to:
main2:4     2015/04/13
  compilers: clang, gcc
  systems contributed to:
main2:5     2015/04/13
compilers: gcc, clang, valgrind
systems contributed to:

main2:6     2015/04/13
compilers: GCC
VisualC++
VisualDSP++
systems contributed to:

main2:7     2015/04/14
compilers: GCC, Clang, MSVC, Valgrind and derivatives.
systems contributed to:

main2:8     2015/04/14
compilers: gcc
clang
icc (Intel)
pgcc (Portland Group)
xlc (IBM)

I used to use −malign-double on i386.
systems contributed to: Cactus (www.cactuscode.org)
Einstein Toolkit (einsteintoolkit.org)
other scientific codes (physics, solving PDEs)
high-performance computing

main2:9     2015/04/14
compilers: GCC/g++, clang/clang++, msvc9, msvc13
No optimization flags except those that encourage using hardware instructions (e.g. −mpopcnt)
systems contributed to:

main2:10    2015/04/14
compilers: GCC and Clang.

I convinced Chrome to keep using nostrictaliasing because it has no perf win on Chrome and is misunderstood.

I use the sanitizer extensively.
I add code randomization and CFI (see pcc’s recent Commits on this).
systems contributed to: Chrome LLVM

main2:11    2015/04/14
compilers: clang
systems contributed to: llvm

main2:12    2015/04/14
compilers: GCC
clang
LLVM core optimisations
systems contributed to:

main2:13    2015/04/14
compilers: GCC
LLVM/Clang
systems contributed to: LLVM

main2:14    2015/04/14
compilers: Gcc, clang, llvm,
systems contributed to:

main2:15    2015/04/15
compilers: gcc
clang
systems contributed to: linux
lighttpd

main2:16  2015/04/15
compilers: gcc (old versions)
Clang
A little experience with coverity
systems contributed to: FreeBSD
CheriBSD
A couple minor clang patches

main2:17  2015/04/15
compilers: gcc, clang, sun C compiler
systems contributed to:

main2:18  2015/04/15
compilers: gcc, clang
systems contributed to: FreeBSD

main2:19  2015/04/15
compilers: gcc, mostly; a little bit of clang and MSVC
systems contributed to:

main2:20  2015/04/15
compilers: gcc, clang, xlc
I no longer remember which flags we use as it’s been different for every gig. Mostly I’m interested in enabling all the warnings.
systems contributed to: I’ve contributed to FreeBSD. I’ve written code for Linux but nothing that was submitted as a patch, just work-related.

main2:21  2015/04/15
compilers: gcc, clang, lint, klocwork
Very many Inhouse option sets in test suites
systems contributed to: freebsd

main2:22  2015/04/15
compilers: Compilers: GCC and Clang
Analysis tools: Clang analyser, Coverity, Tartan

Standard list of compiler flags I use:

systems contributed to: GNOME (http://gnome.org/)

Many systems based on its technologies

main2:23  2015/04/15
compilers: gcc
clang
icc
scan-build
valgrind suite
-fno-red-zone in kernel code to stop stack corruption on x86_64
-fno-strict-aliasing for older code that casts pointers

systems contributed to:

main2:24 2015/04/15
 compilers: gcc
 systems contributed to: 

main2:25 2015/04/15
 compilers: gcc
 systems contributed to: Mostly proprietary

main2:26 2015/04/15
 compilers: gcc, clang, vc++
 systems contributed to: 

main2:27 2015/04/15
 compilers: I primarily use gcc, both as a compiler for C applications and as a
target for HLL compilers. I have also used clang and several platform-specific
commercial compilers over the years, though I avoid closed or platform-tied com
pilers where possible.

For programs targeting C, I use almost exclusively -Wall -O2, plus any platform-
specific flags necessary for correctness (-nostdlibs, -nodefaultlibs, etc.). Th
is ensures sufficient optimization for the compiler to complain about most commo
n errors, but does not get it into trouble on most versions of gcc.

For compilers targeting C, I use -fno-strict-aliasing -fno-optimize-sibling-call
s -fomit-frame-pointer -fomit-leaf-frame-pointer -falign-loops and possibly othe
r optimizations, along with -O3 or -O6 (and -Werror during development, but not
deployment). While these optimizations can make debugging difficult, one does n
ot normally use a C-level debugger to debug the output of another compiler unless
one is the actual compiler vendor, and these optimizations have measurable effe
cts on performance.

main2:28 2015/04/15
 compilers: gcc
 ms visual c++
 clang
 icc

gcc flag: -fno-omit-frame-pointer for debugging and interop purposes

systems contributed to:

main2:29 2015/04/15
 compilers: Gobs. The ones I recall: gcc, g++, xlc, xlC, clang, pcc, dec’s C co
piler (for both mips, vax and alpha), sgi’s C compiler (mips), Intel’s C compil
er, boreland C/C++, TOPS-20 cc (the Greg Titus one), sun’s C compilers, etc

Flags get turned on when code breaks with all the optimizations turned on. The l
ist I’ve had to turn on over the years is too large to hold in my brain.

systems contributed to: FreeBSD

main2:30 2015/04/15
 compilers: gcc, MSVC
 systems contributed to: 

main2:31 2015/04/15
 compilers: 
 systems contributed to: 

14/145 Wednesday March 09, 2016
In the past I have used the BSD Unix C compiler, Sun Solaris C compiler, DEC Ultra and VMS C compilers.

- In particular care taken to catch -Wstrict-overflow in many utilities which are poorly written and the optimiser causes (programmer unintentional) behavior in which the calculation doesn’t wrap.

- never used llvm, gcc is a compiler hacker’s compiler

- various small apps

- I was on the Mercury compiler team which used targetted C as a back-end.

- Microsoft static analysis tools, SDL, prefast, prefix, ESPx, etc.

- My primary development platform is Microchip PIC embedded microcontrollers.
compilers:
systems contributed to:

main2:44 2015/04/16

compilers: GCC
clang/llvm

systems contributed to:

main2:45 2015/04/17

compilers: intel, ms, gnu, pgi,

systems contributed to:

main2:46 2015/04/17

compilers: Gcc, LLVM/Clang, MSVC, Intel C/C++ compiler

systems contributed to: Occasional linux kernel, library, application development mostly on existing open source works

main2:47 2015/04/17

compilers: 
systems contributed to:

main2:48 2015/04/17

compilers: Clang, GCC, Intel C, IBM XLC, Cray C

I often specify the language variant I want because, until recently, few compilers defaulted to C99, which I use because of flexible declarations "for (int i=0; i<n; i++)" especially) and restrict. I also use C11 atomics, hence specify C11 support when necessary. I also _assume_ the C11 memory model, i.e. thread-safe code generation by default, but know of no compiler that violates this (thanks to the ubiquity of Pthread) and no flags to control it (they exist for Fortran compilers...).

systems contributed to: I don’t full understand the question.

I write communication middleware and other libraries for parallel computing. You likely don’t know any of my projects but you can verify via https://github.com/jeffhammond/.

main2:49 2015/04/17

compilers: icc, gcc, clang.with −02 or −03.

systems contributed to: Julia implementation

main2:50 2015/04/17

compilers: I currently mostly use gcc/g++, although I’ve also used clang often enough (I often kept switching between the two after running into some annoyances with one or the other).

I always compile with -fno-strict-aliasing -fwrapv since I already live in enough fear of the compiler "cleverly" miscompiling my code.

systems contributed to: In-house firmware and software, typically interacting directly with hardware (memory-mapped I/O).

main2:51 2015/04/17

compilers: gcc, clang+llvm. Usual −0 flags (typically −0s, −02, −03).

systems contributed to:

main2:52 2015/04/17

compilers: 
systems contributed to:

main2:53 2015/04/17

compilers: gcc clang

systems contributed to: JTC1/SC22/WG14, ca 1999

main2:54 2015/04/17
compilers: clang, gcc
systems contributed to:

main2:55 2015/04/17
compilers: clang, gcc.
systems contributed to: Not possible."

main2:56 2015/04/17
compilers: gcc, clang
systems contributed to:

main2:57 2015/04/17
compilers:
systems contributed to:

main2:58 2015/04/17
compilers:
systems contributed to:

main2:59 2015/04/17
compilers: gcc
clang

-02 is the usual compiler optimization flag
-fno-strict-aliasing because we don’t care to have those bugs, not yet.

  systems contributed to: Google C++ core libraries
Google C++ confidential projects
Google C++ "code janitor" work

main2:60 2015/04/17
compilers: clang, gcc
systems contributed to: Websearch at google

main2:61 2015/04/17
compilers: g++
clang++

Not much experience with C, so take my answers with a grain of salt.
systems contributed to:

main2:62 2015/04/17
compilers: gcc, clang
systems contributed to:

main2:63 2015/04/17
compilers:
systems contributed to:

main2:64 2015/04/17
compilers: gcc, clang, Microsoft Visual C++, plus Xbox, Xbox 360 and PlayStation
3 MSVC, gcc and clang-based toolchains.
systems contributed to: Chrome browser, Chrome for Android, ChromeOS, various games.

main2:65 2015/04/17
compilers:
systems contributed to:

main2:66 2015/04/17
compilers: gcc/g++, MSVC
systems contributed to:

main2:67 2015/04/18
compilers: Clang, gcc, coverity.
systems contributed to: A networking product.
main2:68        2015/04/18
  compilers: gcc, icc, clang
  systems contributed to:

main2:69        2015/04/19
  compilers: Clang, GCC, MSVC, EDG
  systems contributed to: Clang

main2:70        2015/04/20
  compilers: gcc, clang, valgrind, clang-analyzer
  systems contributed to:

main2:71        2015/04/20
  compilers: gcc
  clang
  icc
  systems contributed to:

main2:72        2015/04/20
  compilers: clang
  gcc
  tcc
  msvc
  systems contributed to: "Systems" that you contribute to? What does this mean?
  This question is very ambiguous.

main2:73        2015/04/20
  compilers: gcc
  clang
  systems contributed to:

main2:74        2015/04/20
  compilers: Borland C++Builder, GCC and Clang including all the sanitizers.
  Of course -fno-strict-aliasing, but really, I’ll just say "every single flag GCC has,
or has ever had in any version since 2.7" (it’s not quite true), with a special
  mention to -funsigned-char (pronounced with "fun"!).
  systems contributed to: LLVM mostly, and Clang a bit.

main2:75        2015/04/20
  compilers: clang, icc, gcc, cl
  I generally don’t use particular flags, except -std ones
  systems contributed to:

main2:76        2015/04/20
  compilers: GCC, Clang
  systems contributed to:

main2:77        2015/04/20
  compilers: GCC, clang, msvc
  systems contributed to:

main2:78        2015/04/20
  compilers: 
  systems contributed to:

main2:79        2015/04/20
  compilers: gcc
  clang
  systems contributed to:

main2:80        2015/04/20
Compilers: clang, gcc

Compilation flags depend on what I am going to code. For example, programming with many large integers will make me think about -fwrapv, and programming with lots of bit reuses (e.g. union) around will make me think about -no-strict-aliasing.

systems contributed to:

main2:81 2015/04/20
  compilers: Mostly just GCC C compiler, occasional use of compiler flags, not much to report.
  systems contributed to: All closed source, draconian IP restrictions from past employers.

main2:82 2015/04/20
  compilers: IAR C Compiler
  Mathworks Polyspace
  systems contributed to:

main2:83 2015/04/20
  compilers:
  systems contributed to:

main2:84 2015/04/20
  compilers: GCC, icc, IBM xlc, Solaris Studio cc, clang, llvm, valgrind, AddressSanitizer, ThreadSanitizer, MemorySanitizer, Electric Fence.
  systems contributed to:

main2:85 2015/04/20
  compilers: GCC, Clang, MSVC
  systems contributed to:

main2:86 2015/04/21
  compilers: clang, gcc, MSVC
  systems contributed to: OS X, iOS, Linux, Windows

main2:87 2015/04/21
  compilers: Compilers: icc, gcc, pgicc, mpicc
  Tools: gdb, perf, sep
  systems contributed to:

main2:88 2015/04/21
  compilers: GCC
  systems contributed to:

main2:89 2015/04/21
  compilers: clang
  gcc
tcc
  I use:
  -std to force compilation under different standards for different files, in one case I use different standards in a single project
  -ffast-math, -fomit-frame-pointer for speed
  systems contributed to: FreeBSD

main2:90 2015/04/21
  compilers:
  systems contributed to:

main2:91 2015/04/21
  compilers: GCC, clang
  systems contributed to: GNU coreutils, gnulib

main2:92 2015/04/21
  compilers: gcc, lcc, MSVC, Digital cc, Sun Pro C, Apple MrC
  systems contributed to: various commercial systems, also including the MPS sy
The major project I work on (QEMU) uses -fno-strict-aliasing -- the problems it brings are more painful than any putative benefits, especially given QEMU’s tendency to low-level memory manipulation. If gcc/clang offered a Regehr-style "friendly C" option (http://blog.regehr.org/archives/1180) I would use it in a heartbeat.

Analysis tools:
valgrind, coverity

systems contributed to: QEMU.

Mainly gcc, some clang.
systems contributed to: Exim, BIND, FreeBSD.

Mainly GCC, a bit of MSVC, tiny amounts of Clang (the latter only after checking how the other two optimize).
systems contributed to:

gcc, clang, msvc
-ffast-math -O3
systems contributed to:

Gcc, clang
systems contributed to: OS X, iOS (past)

gnu gcc
intel icc
Microsoft MSVC++
Texas Instruments Code Composer Studio (for DSP chips)
systems contributed to:

GCC, occasionally with -fno-strict-aliasing where it affects particular codes (normally I attempt to avoid things that I know will trigger mis-optimization)
Clang
MSVC
Diab

systems contributed to:

GCC, clang
systems contributed to:
compilers: gcc, often with -fno-strict-aliasing -fno-strict-overflow
systems contributed to: Linux kernel

compilers: GCC
systems contributed to: Minor contributions to GCC backend.

compilers: None
systems contributed to: None

compilers: We currently use gcc and clang. -fno-strict-aliasing is specified when we compile for embedded target hardware because chip-vendor supplied driver code is often horrible, but our code base is tested with strict aliasing enabled (as well as with -fsanitize=undefined).
systems contributed to: The embedded AirStash OS; at a previous employer, compilers for domain-specific languages used for telecommunications systems.

compilers: icc
always used with -intel-extensions to have access to IEEE-754-2008 decimal floats and Cilk Plus
also sometimes used with -parallel for the auto-parallelizer
systems contributed to: dateutils
redland libraries (raptor, rasqal, librdf)
libarchive

compilers: MSVC, Intel, gcc, clang and some compilers for embedded targets that were mostly gcc based.
systems contributed to:

compilers: GCC
systems contributed to:

compilers: GCC, Clang
systems contributed to:

compilers: GCC, clang, msvc, ICC, armcc, cl6x. Long ago there were more, and more particular. Mostly I turn on gnu compatibility and c99, but don’t use many of the c99 syntax features that are incompatible with c++ (the ones not supported by msvc and cl6x)
systems contributed to:

compilers: gcc
clang
systems contributed to:

compilers: GCC, clang, msvc
systems contributed to:

compilers: GCC, clang, msvc, valgrind
systems contributed to:

main2:116  2015/04/21
  compilers: gcc, clang

systems contributed to:

main2:117  2015/04/21
  compilers: gcc, clang

systems contributed to:

main2:118  2015/04/21
  compilers: Compilers
  gcc, clang (address/leak/memory/undefined-behavior sanitizers, static analyzer)

Tools
  perf, gprof, systemtap, google-perftools, valgrind (cache-grind)

  systems contributed to: Linux

main2:119  2015/04/21
  compilers: gcc, g++

  systems contributed to: linux

the python interpreter,
  python C/C++ API

main2:120  2015/04/21
  compilers: GCC, clang

systems contributed to:

main2:121  2015/04/21
  compilers: GCC/Clang (interchangeable)
  -Wall -Wextra -Werror

  systems contributed to:

main2:122  2015/04/21
  compilers:

  systems contributed to:

main2:123  2015/04/21
  compilers: gcc 4.7, clang
  Visual 2012 & 2013

  systems contributed to: Commercial C-code generation products

main2:124  2015/04/21
  compilers: GCC, Microsoft (before Visual, and Visual), Watcom, Borland, IAR, other older compilers for embedded development

  systems contributed to:

main2:125  2015/04/21
  compilers: Clang
  GCC
  -ftree-vectorize tells it to auto vectorize code where easily possible
  -march allows it to optimize for specific processors
  -O2 is the highest level of safe optimization

  systems contributed to:

main2:126  2015/04/21
  compilers: gcc, clang, cl.exe and some variants of these.
I don’t tend to use custom compiler flags to change behavior.

systems contributed to: Unsure what you mean by this.

main2:127 2015/04/21
 Flexelint, Parfait, Coverity, Fortify
 systems contributed to: proprietary

main2:128 2015/04/21
 compilers: 
 systems contributed to:

main2:129 2015/04/21
 compilers: gcc
 clang
 valgrind
 -fno-strict-aliasing : I like aliasing
 -fwrapv : everything I target has two’s complement
 -Ofast : usually don’t care about float precision
 -fomit-frame-pointer : we’re not writing 16-bit code anymore
 -march=native : use all my CPU features, please
 -funroll-loops : usually a small speed boost
 -std=c99 : I always use C99 these days

 systems contributed to:

main2:130 2015/04/21
 compilers: Clang, gcc, coverity
 systems contributed to:

main2:131 2015/04/21
 compilers: 
 systems contributed to: Apple – Mac OS X and iOS.

main2:132 2015/04/21
 compilers: gcc -std=c99
 c51
 gdb
 valgrind
 systems contributed to: 

main2:133 2015/04/21
 compilers: gcc, visual
 systems contributed to:

main2:134 2015/04/21
 compilers: Mainly GCC; also internal compiler tools.
 systems contributed to: No open source contributions. I develop internal language tools that target homebuilt C compilers

main2:135 2015/04/21
 compilers: gcc
 systems contributed to:

main2:136 2015/04/21
 compilers: gcc
 clang
 systems contributed to:

main2:137 2015/04/22
 compilers: gcc
 clang
 vc++
 systems contributed to: 

Wednesday March 09, 2016
compilers: diab, etec, gcc, pclint
systems contributed to:

compilers: gcc, custom analysis tools
commercial: Coverity, CodeSonar
systems contributed to: Commercial products

compilers: gcc, clang, msvc, armcc
systems contributed to:

compilers: GCC, valgrind for pointer/memory correctness, and gdb
systems contributed to: Mostly personal projects.

compilers: gcc
gdb
clang
systems contributed to:

compilers: gcc, clang
systems contributed to: linux

compilers: gcc
codewarrior
composer studio
visual studio
systems contributed to: Automotive ECUs

compilers: gcc clang
systems contributed to:

compilers: gcc, clang, msvc, tcc
systems contributed to:

compilers: CLANG
GCC
(miscellaneous legacy compilers over the years)
systems contributed to:

compilers: Compilers I use: Clang, gcc, icc, solaris studio, icc, xlc
Analysis tools I use: clang analyzer, Coverity, cppcheck, address sanitizer
systems contributed to: Contribute to Linux kernel.

Develop applications for Linux, AiX, Solaris, FreeBSD, OS X, Windows, Android.

compilers: GCC, clang, with the --analyze flag. Always use -Wall
systems contributed to:

compilers: gcc, custom analysis tools
commercial: Coverity, CodeSonar
systems contributed to: Commercial products

compilers: gcc, clang, msvc, armcc
systems contributed to:

compilers: GCC, valgrind for pointer/memory correctness, and gdb
systems contributed to: Mostly personal projects.

compilers: gcc
gdb
clang
systems contributed to:

compilers: gcc, clang
systems contributed to: linux

compilers: gcc clang
systems contributed to:

compilers: gcc, clang, msvc, tcc
systems contributed to:

compilers: CLANG
GCC
(miscellaneous legacy compilers over the years)
systems contributed to:
compilers:
systems contributed to:

main2:152       2015/04/23
  compilers: gcc, clang, gdb, lldb, valgrind
  systems contributed to: Embedded Linux, application software for Linux and Windows with Qt and C++

main2:153       2015/04/23
  compilers: gcc, clang, msvc, lint, pc lint, splint, oclint
  systems contributed to:

main2:154       2015/04/23
  compilers: gcc, clang
  systems contributed to:

main2:155       2015/04/23
  compilers: clang
gcc
  systems contributed to:

main2:156       2015/04/24
  compilers: gcc, clang, msvc; asan and valgrind; familiar-with-existence of tons of static tools but rarely use them.
  systems contributed to: too private, sorry

main2:157       2015/04/24
  compilers: gcc clang
  systems contributed to: mailfront ezmlm-idx darktable luminance-hdr

main2:158       2015/04/24
  compilers: Compilers going back 30 years: THINK C, MPW C, Borland C and C++, Microsoft C++ 7.0 and Visual C++ versions, CodeWarrior C++, gcc to some extent, clang to some extent, TI Code Composer Studio C compiler, Understand for C/C++, etc.
  systems contributed to:

main2:159       2015/04/24
  compilers:
  systems contributed to:

main2:160       2015/04/24
  compilers: I’ve used gcc (off and on) for approximately 25 years. Usually I use more-or-less default optimization flags, although I have worked on code bases (older versions of the Python implementation) that required no-strict-aliasing (because the old implementation of inheritance for extension types violated aliasing requirements).

Also, some of the code I wrote 15-20 years ago assumed that signed integer overflow would behave "as expected" using 32-bit two’s-complement; if I were to try to port that code to a modern compiler, I’d probably use −fwrapv rather than try to fix the code.

I’ve used gcc mostly for Linux/Unix development (for SPARC, 32-bit x86, and ARM), but also for embedded development (for Atmel AVR, for ARM, and for Motorola 68332 (which is between a 68010 and a 68020, basically)).

I’ve also used a few non-gcc-based C compilers occasionally (the SunOS and Ultron x cc, Apple’s clang-based iOS compiler).

systems contributed to: Most of the code I’ve written has been proprietary, but I’ve contributed significant amounts of code to the Sage computer algebra system, some of which was in C.

main2:161       2015/04/24
  compilers: gcc
clang
clang static analysis
astree
  systems contributed to:

main2:162  2015/04/26
  compilers: clang
gcc
  systems contributed to:

main2:163  2015/04/26
  compilers: gcc
msvc
CodeWarrior
  systems contributed to:  Game development

main2:164  2015/04/26
  compilers: gcc, clang, icc, msvc
  systems contributed to:

main2:165  2015/04/27
  compilers: GCC, Clang, Valgrind.
  systems contributed to:  MATLAB and Simulink code generation (at MathWorks, Inc.).

main2:166  2015/04/30
  compilers: gcc
  systems contributed to:

main2:167  2015/04/30
  compilers: 
  systems contributed to:

main2:168  2015/04/30
  compilers: GCC
  systems contributed to:

main2:169  2015/05/02
  compilers: GCC, Clang, QNX forks of GCC toolchain
  systems contributed to:  Worked on BlackBerry10 in 2013.

main2:170  2015/05/02
  compilers: 
  systems contributed to:

main2:171  2015/05/06
  compilers: 
  systems contributed to:

main2:172  2015/05/07
  compilers: gcc
  llvm
compCert
compCertTSO
  systems contributed to:

main2:173  2015/05/15
  compilers: GCC, Microchip C30, (Cl8 for PIC microcontrollers, although it is not real C)
  systems contributed to:  microcontroller code (in C), application on PC (mainly in C++)

main2:174  2015/05/15
  compilers: gcc
  llvm
no-strict-aliasing for using a C compiler as a high-level compiler backend standalone for declaring globals with the same name as standard library functions, but a different type, in a high-level compiler backend

systems contributed to:

main2:175 2015/05/22
  compilers: gcc, hi-tech c
  systems contributed to:

main2:176 2015/05/22
  compilers: clang
  gcc
  xlc -qstrict (to make kahan summation work)
  systems contributed to: pp, Polly

main2:177 2015/05/31
  compilers: gcc, clang
  systems contributed to: networked multimedia transport

freebsd:0 2015/04/25
  compilers: gcc, clang, keil (for 8051-based embedded systems), various other embedded compilers.
  systems contributed to: FreeBSD (emphasis on de, ZFS, xnb)

freebsd:1 2015/04/25
  compilers: GCC, Clang, TenDRA, Coverity Prevent, various minor static analysis tools.

Rather than controlling optimization/behaviour, I tend to rely upon extensive compiler warnings, e.g. with Clang using -Weverything and only turning off a small class of things (e.g. -Wpadded.)
  systems contributed to: FreeBSD

freebsd:2 2015/04/25
  compilers: gcc
  clang
  splint
  systems contributed to: Linux
  FreeBSD
  RTEMS

freebsd:3 2015/04/25
  compilers: clang, gcc
  systems contributed to: FreeBSD

freebsd:4 2015/04/26
  compilers: gcc
  systems contributed to: FreeBSD

freebsd:5 2015/04/26
  compilers: clang, gcc 4.2.1

Current only use various -m to turn on various intrinsics necessary for optimizations.

I do know that the FreeBSD kernel has turned on wrapv (unsigned wrap being defined instead of undefined), but others are better to discuss that.
  systems contributed to: FreeBSD, specifically crypto code, but have contributed to other parts of FreeBSD

freebsd:6 2015/05/14
  compilers: gcc, clang
  systems contributed to: FreeBSD

freebsd:7 2015/04/26
  compilers: gcc
clang
Clang static analyzer

systems contributed to: FreeBSD
OpenAFS

freebsd:8 2015/04/28
compilers: I am familiar with clang and gcc.
Sometimes we use -fwrapv to get defined twos-complement signed integer overflow,
and sometimes we use -fno-strict-aliasing.
systems contributed to: MIT Kerberos
OpenAFS
FreeBSD

gcc:0 2015/04/17
compilers: GCC
We use no-strict-aliasing when building the OpenJDK Java VM: there is so much poor
inter casting that nobody wants to go through the million lines or so to fix it all.
systems contributed to: OpenJDK.

gcc:1 2015/04/17
compilers: GCC, many options.
systems contributed to: Go, GCC, GNU binutils, etc.

gcc:2 2015/04/18
compilers: gcc, pcc, libfirm/cparser
__attribute__((__may_alias__)) for accessing representations as words.
systems contributed to: musl libc (maintainer), glibc (issue reporting and de
bugging, etc.), gcc (bug reports), Linux (bugs), pcc (bugs), ...

gcc:3 2015/04/21
compilers: gcc, clang, MSVC
systems contributed to:

gcc:4 2015/04/23
compilers: GCC, -fno-expensive-optimisations sometimes to work around broken c
ode in benchmarks
systems contributed to: GCC, gas, newlib

googcomp:0 2015/05/26
compilers: clang, clang-tidy
systems contributed to:
goog:0 2015/05/26
compilers: GCC, LLVM clang, TCC
systems contributed to:
goog:1 2015/05/26
compilers: gcc
cc (solaris)
lint
lex
yacc
systems contributed to:
goog:2 2015/05/26
compilers: gcc
systems contributed to:
compilers: gcc...?
systems contributed to: not really possible

compilers: gcc, clang, msvc. I usually restrict myself to standard -On flags and whatever build team / release group deems optimum, such as as using FDO.
systems contributed to:

compilers: gcc, PGI, Sun Pro compilers, Intel compilers, IBM XLC
I always turn on all possible warnings
systems contributed to:

compilers: gcc
clang
systems contributed to:

compilers: gcc
clang
no-strict-aliasing (because we think it would cause bugs)
no-exceptions (to save time and codegen size)
systems contributed to: primarily code used for backend systems at Google.

compilers: gcc
g++
valgrind
gdb
systems contributed to: Google internal code, both for server and Android

compilers: Bcc clang msvc
systems contributed to:

compilers: gcc
systems contributed to: linux, freebsd

compilers: I’m familiar with gcc, clang, and MSVC, although I couldn’t tell you where the gcc and clang differ.

I’ve been known to fiddle with auto-vectorization and tree optimization flags if the code looks particularly amenable to it, but typically it ends up not being worth the effort, so I don’t have any specific flags that I keep in mind.
systems contributed to:

compilers: gcc
clang
cl.exe
systems contributed to:

compilers: gcc, clang, icc
systems contributed to: Linux

compilers: gcc
clang
asan
purify

systems contributed to:

google:15       2015/05/26
compilers: gcc, clang

systems contributed to:

google:16       2015/05/26
compilers:
systems contributed to:

google:17       2015/05/26
compilers: gcc clang

systems contributed to:

google:18       2015/05/26
compilers: gcc, clang

systems contributed to:

google:19       2015/05/26
compilers: gcc, whatever Microsoft Visual studio had in last 6 years, various compilers for embedded systems
systems contributed to: Not at liberty to say this

systems contributed to:

google:20       2015/05/26
compilers: gcc, clang

systems contributed to:

google:21       2015/05/26
compilers: GCC, Clang; I do not use a consistent pallet of flags, apart from warning flags, which I usually pile on, including -pedantic.
systems contributed to:

google:22       2015/05/26
compilers: gcc, valgrind, clang

systems contributed to:

google:23       2015/05/26
compilers: gcc, llvm, cl

systems contributed to:

google:24       2015/05/26
compilers: GCC, ICC

systems contributed to:

google:25       2015/05/26
compilers:
systems contributed to:

google:26       2015/05/26
compilers:
systems contributed to:

google:27       2015/05/26
compilers: gcc

systems contributed to:

google:28       2015/05/26
compilers: gcc

systems contributed to: linux

google:29       2015/05/26
compilers: gcc, usually with -03

systems contributed to:
compilers: GCC
systems contributed to: Not an expert in the slightest, but I dabble.

compilers: gcc
MSVC
systems contributed to:

compilers: gcc -- Compiler used for most development work.

Much less familiarity with a few others: Sun and SGI’s compilers, icc, and clang

compilers: gcc, cppcheck, clang, coccinelle
systems contributed to: Linux, U-Boot

compilers: gcc, clang, Amsterdam Compiler Kit
systems contributed to:

compilers: gcc, clang
systems contributed to:

compilers: gcc, clang
systems contributed to:

compilers: gcc, clang, nvcc, sun cc, visual studio
sparse, lint, coverity scanner
arch-specific tuning (march/mcpu) for instruction emission
-O2 for optimization and dataflow analysis-based warnings
-Wall -Werror -W and many other -Wfoo flags

systems contributed to: linux kernel
sprezzos project (defunct)
omphalos
growlight

compilers: I’ve mostly used Clang and GCC. I’ve used -fno-strict-aliasing in most projects because I’ve worked on. These projects relied on casting between different struct pointer types to implement object systems, which I think is undefined. I’m hesitant to go beyond -O2 for similar reasons.

compilers: In grad school, I wrote some transformation and analysis tools for Clang. Professionally, I’ve worked on V8 (JavaScript obviously being very different than C, but compiler internals are structurally similar to Clang). I also worked on performance tools for BrewMP (operating system for feature phones), which was mostly written in C.
Mostly developing firmware and using options like -ffreestanding, -nostdlib, -nostdinc, -fno-common and -fomit-frame-pointer for that. Also dealing with linker scripts a lot and a big fan of -Wl,--gc-sections -fdata-sections -ffunction-sections.

systems contributed to: coreboot, Linux
compilers: for personal projects i mostly use

gcc -Wall -pedantic -std=c99 -D_POSIX_C_SOURCE=200809L

i use -fexcess-precision=standard -frounding-math to get reasonable floating-point semantics when working on mathematical functions (gcc can still miscompile code that has fenv access).

i also used cpars/firm and pcc for specific projects where i had to modify the implementation (they are easier to modify than gcc or clang).

i used the clang static analysis tool as well.

systems contributed to: musl-libc, glibc, gcc

libc:4 2015/05/27
  compilers: gcc primarily, sometimes Clang (particularly for checking warnings)


systems contributed to: Debian GNU/Linux. I used to do a lot of work on OpenAFS and Kerberos, although not as much at the moment. In the past, I used Solaris heavily.

libc:5 2015/05/27
  compilers: gcc
  clang
  systems contributed to:

libc:6 2015/05/27
  compilers: gcc
  systems contributed to:

libc:7 2015/06/05
  compilers: GCC
  MSVC

Clang (both as a compiler, and for static analysis)
Coverity for static analysis
systems contributed to:

llvm:0 2015/04/23
  compilers: gcc
  systems contributed to: FreeBSD
Proprietary embedded system.

llvm:1 2015/04/23
  compilers: gcc
  systems contributed to:

llvm:2 2015/04/23
  compilers: I use Clang, GCC, and MSVC regularly. The most important flags we use are probably -fno-exceptions and -fno-rtti, to eliminate the cost of C++ features that we don’t use.
  systems contributed to: Clang and LLVM

llvm:3 2015/04/24
  compilers: MSVC clang gcc
  systems contributed to: see toomaytech.com

llvm:4 2015/04/24
Open source contributions have been minimal and very scattered.
regehr-blog:10 2015/04/22
compilers: gcc, hi-tech/microchip C for my sins
systems contributed to:

regehr-blog:11 2015/04/23
compilers:
systems contributed to:

regehr-blog:12 2015/04/23
compilers: Clang
gcc
Valgrind
MSan
ASan
systems contributed to: Currently, Google Street View camera systems (i.e., the code that actually runs on vehicle to control and log data from cameras and other sensors on Google’s Street View cars). I do more work in C++ than C at this job, though previous jobs have been strictly C.

regehr-blog:13 2015/04/23
compilers: gcc
systems contributed to:

regehr-blog:14 2015/04/23
compilers: gcc
PC-lint
systems contributed to: Embedded systems for avionics. Using no operating system, or VxWorks, or Integrity-178

regehr-blog:15 2015/04/23
compilers: gcc
clang/LLVM
flexelint
coverity
Some proprietary C compilers for embedded systems
systems contributed to: Proprietary systems in mobile telecom.

regehr-blog:16 2015/04/23
compilers:
systems contributed to:

regehr-blog:17 2015/04/23
compilers: GCC
Clang/LLVM

systems contributed to: Personal projects
An unreleased, proprietary Linux kernel module

regehr-blog:18 2015/04/24
compilers: gcc, no special flags

Intel C/C++ compiler in some HPC codes it benefits from -ansi-alias since it makes stricter assumptions to restrict/__restrict pointers
systems contributed to:

regehr-blog:19 2015/04/24
compilers: gcc, clang
systems contributed to: Linux

regehr-blog:20 2015/04/24
compilers: GCC

I’ve previously used a lot of other compilers, but not in the last 10 years. Microsoft Visual C++ 1.0 and a few subsequent versions
MetaWare
Whitesmith
A few compilers from Unix vendors (e.g. IBM’s xlcc).
EDG
Borland C/C++ 3.1, Turbo C

systems contributed to:  Lots of GNU software:
findutils
coreutils
gnulib
(slightly) glibc

Some other free software (e.g. WINE, Gnome) but not recently.

I also contribute to back-end and library code at a major technology company with a very large code base, but I’m not willing here to state which one.

regehr-blog:21  2015/04/25
  compilers: gcc, clang, msvc

regehr-blog:22  2015/05/01
  compilers: GCC
Clang

regehr-blog:23  2015/05/02
  compilers: GCC, Intel C, SGI Irix C, AIX C, Purify, etc.
systems contributed to:  Lots of free software.

regehr-blog:24  2015/05/05
  compilers: GCC, Clang/LLVM, MS Visual C(++)
systems contributed to:  In-house libraries and products

regehr-blog:25  2015/05/26
  compilers: gcc, microchip c18
  −fno-move-loop-invariants (to improve size optimisation)
    (also __attribute__((naked)); __attribute__ ((noreturn)) )

warnings, because new compiler versions break old code:
Wunsafe-loop-optimisations
Wstrict-aliasing

systems contributed to:

regehr-blog:26  2015/05/26
  compilers:
systems contributed to:

regehr-blog:27  2015/06/02
  compilers: gcc
clang
systems contributed to:

regehr-blog:28  2015/06/03
  compilers: gcc

In the Gforth build, we use −fno-gcse −fcaller-saves −fno-defer-pop −fno-inline
−fwrapv −fno-strict-aliasing −fno-cse-follow-jumps −fno-reorder-blocks −fno-reorder-blocks-and-partition −fno-toplevel-reorder −falign-labels=1 −falign-loops=1 −falign-jumps=1
−fwrapv, because we want modulo arithmetics
−fno-strict-aliasing, because we access casted pointers

-falign-labels=1 -falign-loops=1 -falign-jumps=1, because we copy the machine code around and therefore don’t benefit from the padding (and we do our own padding on some platforms).

Systems contributed to: Gforth

regehr-blog:29 2015/06/26
compilers: gcc, icc, msvc
systems contributed to:

regehr-blog:30 2015/08/27
compilers:
systems contributed to:

regehr-blog:31 2015/09/07
compilers: gcc
clang
mingw32-gcc
mingw64-gcc
systems contributed to:

regehr-blog:32 2015/09/07
compilers: gcc
systems contributed to:

regehr-blog:33 2015/09/07
compilers:
systems contributed to:

x11:0 2015/05/26
compilers: compilers: gcc, Oracle Solaris Studio, clang
analysis: Oracle Parfait, Coverity, cppcheck
systems contributed to: X Window System (X.Org)
Oracle Solaris operating system

x11:1 2015/05/26
compilers: gcc
systems contributed to: TeX, X11

x11:2 2015/05/26
compilers: gcc, g++, clang. Familiar with various CFLAGS (I’m a Gentoo Developer).
systems contributed to: Mesa, Gentoo, X11

x11:3 2015/05/27
compilers:
systems contributed to: NetBSD

x11:4 2015/05/27
compilers: gcc, clang, old MSVC
systems contributed to: NetBSD

x11:5 2015/05/27
compilers: gcc, clang, to some extent tendra; clang-static-analyzer (a bit), sparse (not recently though), been looking at frama-c but haven’t done anything with it yet. also, if getting open-source Coverity reports counts, that.

Mucking with compiler flags is unportable and unreliable, and not future-proof. One has to cope with what the compilers decide to do, and hope that whatever insanity next year’s compiler comes up with doesn’t require issuing extensive patches.

Systems contributed to: Of late mostly NetBSD. I also have a teaching OS called OS/161 where much of this stuff runs into undergrads’ programming mistakes.
I have some C tools of my own but none of them have reached the stage of being ready for anyone else to do anything with them.

I should note here that NetBSD being NetBSD we take these issues fairly seriously; we’ve seriously discussed doing a port to a 36-bit architecture (though it hasn’t happened yet) specifically to shake out code problems, and if a credible DS 9000-type platform appeared we’d certainly do a port to it.

(Given that you’re doing this survey, I assume you recognize "DS9000", and if not you need to look it up.)

btw, if you want to quote anything, contact me and I can probably produce a pithier version. And, as I’m a researcher working on somewhat related material, feel free to contact me for further discussion.

x11:6 2015/05/30
    compilers: gcc clang valgrind
    systems contributed to:

xen:0 2015/05/26
    compilers: gcc
    MSVC
    systems contributed to: Xen hypervisor
    Xen Windows PV drivers

xen:1 2015/05/27
    compilers: gcc, clang
    -ffunction-sections for easy objectdiffing
    Also, C++-specific options to disable certain language features when doing OS work (-fno-exceptions, -fno-rtti)
    systems contributed to: L4/Fiasco.OC, L4Re, more recently AWS infrastructure work

xen:2 2015/05/27
    compilers: gcc
    systems contributed to: Xen
    Linux

xen:3 2015/06/03
    compilers: GCC mainly.
    Have experimented with various analysis tools, Coverity primearily.
    systems contributed to: Xen x86 hypervisor maintainer
    XenServer Ring0 developer

========================================================
===  MAIN QUESTION RESPONSES  ===
========================================================

[1/15] How predictable are reads from padding bytes?
If you zero all bytes of a struct and then write some of its members, do reads of the padding return zero? (e.g. for a bytewise CAS or hash of the struct, or to know that no security-relevant data has leaked into them.)

Will that work in normal C compilers?

yes : 116 (36%)
only sometimes : 95 (29%)
no : 21 (6%)
don’t know : 82 (25%)
I don’t know what the question is asking : 3 (0%)
no response : 6

Do you know of real code that relies on it?

yes : 46 (14%)
yes, but it shouldn’t : 31 (9%)
no, but there might well be : 158 (49%)
no, that would be crazy : 58 (18%)
don’t know : 25 (7%)
no response : 5

If it won’t always work, is that because [check all that apply]:
- you’ve observed compilers write junk into padding bytes : 31 (20%)
- you think compilers will assume that padding bytes contain unspecified values and optimise away those reads : 120 (79%)
- other : 150

- A clever compiler could use padding bytes of a struct on the stack as temporary storage. I don’t know if any do this though,
- Reads MUST mask out the padding, but the compiler might be allowed to overwrite for write-width reasons. (=> 8-bit variable, but architecture supports only 32-bit writes)
- see comment
- Probably illegal optimization. But wouldn’t be surprised if it was exploited because who uses padding bytes, really?
- I wrote a paper about this
- An int16 with padding next to it could legitimately turn into an int32 write pasting junk from the register’s other word if that’s convenient for your arch
- It depends how the zeros were written
- I haven’t observed it but I’m sure some compilers will write junk into padding bytes
- Standard may not specify
- you’ve observed compilers write junk into padding bytes, you think compilers will assume that padding bytes contain unspecified values and optimise away those reads, Compiler writes like to punk you.
- you think compilers will assume that padding bytes contain unspecified values and optimise away those reads, Compiler is free to overwrite padding
- It’s surely undefined behaviour.
- The struct is stored in contiguous memory and writing values to a member larger than the allocated member size bleed over to following members
- you think compilers will assume that padding bytes contain unspecified values and optimise away those reads, I assume future compilers will write junk into padding for performance reasons
- if padding bytes are never read, all writes are dead stores and compilers are free to remove them. need volatile memset.
- you’ve observed compilers write junk into padding bytes, you think compilers will assume that padding bytes contain unspecified values and optimise away those reads, Contents of padding bytes are implementation defined
- you think compilers will assume that padding bytes contain unspecified values and optimise away those reads, compiler might optimize away the zeroing if the value is not read before being written again
- you’ve observed compilers write junk into padding bytes, A cache line copy or DMA transfer could result junk in the padding
- Standard doesn’t require it, I’ve heard some compilers optimise it, I don’t remember the exact optimisations.
- you think compilers will assume that padding bytes contain unspecified values and optimise away those reads, writes can be optimized out too
- I assume compilers will write junk, but I’ve not observed it
- idk
- you think compilers will assume that padding bytes contain unspecified values and optimise away those reads, a write might use a wider store instruction than needed, if it’s faster or more convenient, which might overlap with the padding
- I think it’ll work, but it seems like a dumb thing to rely on
- compilers can assume that padding bytes contain unspecified values and may write to them as part of writing some other part of the structure.
- you think compilers will assume that padding bytes contain unspecified values and optimise away those reads, Compilers may not write to padding bytes
- Will not write to padding bytes, so reads could contain previous data
- you’ve observed compilers write junk into padding bytes, I don’t think it is
compliant with the standard to make assumptions about the contents of padding bytes.
- see comment
- you think compilers will assume that padding bytes contain unspecified values and optimise away those reads, Behavior clearly undefined: may work on some compilers, but compiler has the right to optimize it
- Compiler may use large-than-necessary store instruction (e.g. int instead of char)
- compiler is free to store larger type which may have garbage in bits that overlay padding
  - I think that on some targets the compiler won’t bother to read and mask when a large write would otherwise modify padding bytes.
  - I imagine the compiler might write junk to padding bytes, but I’ve never bothered to check (let alone observe) this behavior
- Padding at the end of the struct stored in volatile memory may contain data from other segments on some systems
- An optimizing compiler might write more data than necessary, knowing the padding doesn’t matter. For example to quickly initialize 3 "short"s that are padded out from 48 to 64 bits.
- compilers do weird things, I haven’t observed it, but I wouldn’t be surprised
- you think compilers will assume that padding bytes contain unspecified values and optimise away those reads, Compilers may assume padding bytes contain unspecified values and may make wider writes as an optimization (e.g. a 64-bit write into a 32-bit member from a 64-bit source variable)
- you think compilers will assume that padding bytes contain unspecified values and optimise away those reads, I expect compilers to write junk into padding bytes on "weird" platforms, but haven’t observed it.
- I wouldn’t assume padding bytes would be present or that there would be a consistent amount of padding on different machines.
- you think compilers will assume that padding bytes contain unspecified values and optimise away those reads, bit fields
  - I don’t believe that the standard guarantees it.
  - It isn’t well defined behavior by the standards, so compilers can do what they want with it.
  - What reads???
- *writes* to members of the struct may freely overwrite padding
- you think compilers will assume that padding bytes contain unspecified values and optimise away those reads, I think compilers are *permitted* to write junk into the padding bytes (although I have not observed it)
- Given struct A { uint8_t a; uint8_t b; uint64_t c; }, I expect some compiler to optimize "s.a = 5; s.b = 10;" into something like "mov ax, 0x0a05; mov [s], eax", depending on instruction encoding sizes, alignment requirements, whether a particular RISC chip has 32-bit constant loads or 16-bit stores, etc.
- Compiler allowed to corrupt padding if that achieves faster code (by writing overlarge type).
- Elided stores to padding bytes
- See comment
- Smells of undefined behaviour, but unsure
- you think compilers will assume that padding bytes contain unspecified values and optimise away those reads, padding might not be copied when the struct is copied
  - see below
- you think compilers will assume that padding bytes contain unspecified values and optimise away those reads, the zeroing may be optimized

Comment
  I’d assume that reading those bytes would be an undefined behavior and at aggressive optimization levels compiler would interpret in a way that benefits optimization. (main2.2:7)
- Sometimes, a struct accessed in one thread or core is a subset of a more populated struct used from another thread/core. Touching padding is quite illegal, it might touch valid data seen by others. (main2.2:8)
- I would expect this code to work:

```c
struct foo
```
I don’t think there’s a way for me to write a structure so that the compiler can’t add random padding anywhere it likes, anyway. So potentially every structure has padding bytes.

If the compiler can write random values into padding bytes, then you can never treat a structure as an array of bytes when comparing or hashing, which seem like normal things to want to do.

The same applies for hashing. (main2:9)
- if the struct was aliased to a char * or similar and zeroed manually I’d expect modern compilers to potentially do weird things.

I’d be very surprised if a compiler optimized out a memset() but I could see it happening if the same function set most of the values. Doing so would be quite a terrible optimization if the struct ever hit disk. (main2:16)
- If a structure has a byte followed by the rest of a word worth of padding, it may be faster to perform a full-word write of a register to memory...

I’m pretty sure that x86 is safe though. (main2:18)
- I don’t know what the spec says (I kind of suspect it says it’s undefined), but there’s so much code which relies on this that no sane compiler could ever do anything different. (main2:19)
- I know compiler writes have a lot of flexibility and I don’t recall if the standard addresses this. A naive implementation would leave padding bytes alone. (main2:20)
- Under the aliasing rules, you might not even see the non-padding bytes correctly if you’re not careful. (main2:23)
- I have not *observed* this but, thinking about it, I believe it could happen. (main2:28)
- Generally, this will work. However, the standard makes no statements about the values of the padding bytes, only that certain bytes may be added for alignment purposes. Without specific guarantees to the contrary, compiler writers are free to generate code that might, as a side effect, alter the padding bytes if that will result in more efficient code.

Also, I object to "normal" C compilers. What’s normal anyway. They are all pathologically deranged in ways that most folks wouldn’t call normal. (main2:29)
- Assuming "zeroing a struct" means using memset() on block of memory struct is in, then padding should read as zero. (main2:30)
- Compiler may optimise writes to use full words, overwriting the padding. (main2:31)
Hmm, I guess this depends on whether you assign zero to each field or cast the pointer to a char * then treat it as a vector of size sizeof(T) for zeroing. Then I would expect to see zeroes everywhere. (main2:35)

This property is essential for systems code. In fact this is related to the fact that ARM processors had to go back and add 16-bit writes because of the systems and concurrency and device driver code that simply had to have a write to a field update exactly that field. (main2:37)

For example structures copied from the userland must not leak data. (main2:38)

This behaviour is similar to a union of two structs, how one member can overlap another in RAM. It is expected and the programmer should anticipate this. (main2:40)

I don’t know for sure whether code relies on it, but it seems likely to me and I would expect to be able to rely on it.

An attribute to be able to explicitly state one’s expectations would be preferred of course. (main2:50)

How are you zeroing all the bytes? memset ought to zero the padding too. Doing a bunch of field assignments shouldn’t zero them. I don’t know what initializing to a zero value does. (main2:55)

"only sometimes" == sometimes it just happens to work, or may be a register happened to have zeroes in higher bits. (main2:64)

Nonzero padding is sometimes clobbered by zeroes in some real compilers, but writing nonzero bits (that were not there before) is unlikely to happen. (main2:69)

I assume it would work in ‘normal’ compilers. But also assume that it shouldn’t be relied upon. IF the standard doesn’t mention specifics about padding bytes’ storage, that sounds like the kind of thing a compiler may exploit. (main2:72)

IIRC it’s valid to store an object in another object’s padding (pursuant to alignment, etc.). (main2:74)

reading the padding bits results in undefined behaviour. It might work or not, the compiler is free to write to that memory anything he wants. (main2:78)

I often see a related issue in which it’s believed that unions will offer serialization "for free" e.g.:

```c
union {
  struct my_awesome_struct s;
  char serialized[N];
}
```

which relies on implementation-specific treatment of padding bytes, endianness, etc. This is very common, and super bad practice. What’s worse, at least half the advice you read about this sort of thing on popular message boards like SO actually says it’s good (and be heavily upvoted)! The usual rationalization is something like, "well, it’s for an embedded system so I control both sides of the interface" which is lame.

You have not specified how the bytes of the struct are to be zeroed. Is it by memset, calloc, designated initializer, line by line assignment?

I think it is reasonable for the compiler to optimize away the first of a pair of stores if there is no load in between and the memory has not been declared volatile. (main2:83)

If a word contains char or bitfield values, a compiler might reasonably store the whole word (with junk from intermediate calculations in the padding bits) rather than mask and store. (main2:94)

My belief is that this works in a normal compiler, but it’s not something I’ve ever thought to check - I’ve just assumed that it leaves the padding alone. (main2:99)

Code that might (hypothetically) rely on predictable behavior here: naive memcmp of struct objects. (main2:100)

This ought to be controllable with an attribute on the struct type in my opinion. (main2:106)
The cases where it verifiably doesn’t work with icc are bitfields.

- We write structs with padding to disk files and assume the padding bytes will be zero if the struct is memset to 0 and then the members are written. AFAIK we only assume this for testing (checking that files are identical), but don’t rely on it for anything else. (main2:108)
- Code has to work everywhere, on really odd processors. Only do what is sure to work. (main2:111)
- I usually use explicit "padding attributes" in order to pack the struct because of full structure assignment that don’t zero the padding. (main2:117)
- If you want to rely on padding being zeroed, make explicit padding fields (uint8_t arrays). (main2:121)
- Clearly it is not defined and compilers can do what they want. I know that valgrind can’t complain about uninitialized reads until they contribute to control flow or data flow in some way, partially due to this reason. (main2:123)
- I think it should work as long as the reads and writes are done while the struct is aliased through char*.

Assuming that "zero all bytes" means either directly:

struct foo myfoo = {0};

or using memset:

struct foo myfoo;
memset(&myfoo, 0, sizeof(myfoo));

Explicitly not element-by-element zeroing.

struct foo myfoo;
myfoo.a = 0;
...
myfoo.z = 0;

code that relies on it is of the form

myfoo = thyfoo;
memcpy(&thyfoo, &myfoo, sizeof(myfoo));

Assuming here that you are zeroing all bytes of a struct with memset or similar. (main2:134)

- I would hope that compilers wouldn’t optimize out zero padding, though I haven’t looked into the matter. (main2:141)
- I am not aware of anything being specified in the C Standard. I am guessing the compiler only write the necessary field length when writing to a struct (as doing different would write in adjacent fields).

This is why it should work in most cases. Having said that, there may be optimizers that take advantage the the neighbour field is padding and use some other instruction to write the data (therefore overwrite the padding). (main2:144)

- Assume the idiom of memset the entire struct. It seems like the compiler can always marshall the data when needed (for a fun call). But then probably local variables aren’t valid and the compiler probably doesn’t deal with data races (multitasking). (main2:147)
- bit masking packed word size structures usually relies on padding to be a certain value. For example, performing SWAR on a compressed fixed point quaternion stored in a uint32_t or uint64_t. (main2:148)
- Don’t have padding (main2:151)
- I have no idea if it’s legit spec-wise or even if it’s compiler-invariant; I have seen people hashing structs -- with padding -- that they memset though. I wouldn’t be surprised to see it not-work. (main2:154)
- The question is confusing and has some jargon in it; what does "bytewise CAS" mean? I think this has to do with RAM, but the more common meaning I know is Chemical Abstract Service. I don’t think the question should use jargon like this that is will not necessarily be understood by the target audience. (main2:158)
- It’s silly to optimise away writes to padding. Small structures should already be appropriately memory aligned and hence not writing to padding doesn’t improve cache performance. Large structures already exhibit poor(er) cache performance (and are symptomatic of poor design in my opinion). Given that, not writing to padding requires unrolled memset, which would be worse for instruction cache p
erformance.

Also, yes we hash these kinds of structures. (main2:163)
- Gil: I guess the compiler might optimise away the zero-initialisation if all the members are written before the padding is read.

Viktor thinks that if the compiler does a wide write, it will use zeros not junk. (main2:172)
- Question is imprecise. By "padding bytes" do you mean bytes explicitly added by the programmer like "char padding[2]" or bytes added by the compiler to reach an alignment boundary? I assume the latter. And by "zero all bytes" do you mean bzero(&x, sizeof(x)) or assigning zero to all members? I assume the former.

In embedded systems, best practice is to manually align structs so the compiler won’t add any padding bytes at all. (freebsd:0)
- I had a hard time answering the first part of this question as my understanding is that it is undefined what the values of the padding bytes are. So, even if you zero all the members, the compiler is free to leave the padding bytes unwritten.

Also, you do not specify how you zero all the bytes of a struct, nor if you would include the padding bytes as part of the struct or not.

I am also a bit unusual in that I prefer to use:
struct foo a = {};
or
a = (struct foo){};

to zero fill my struct. I use this so that things like pointers get properly initialized, as per C standard, the NULL pointer though is equivalent to the integer 0, it’s in memory representation may be different. (freebsd:5)
- Until recently, I would have assumed that doing

```c
bzero(&some_struct, sizeof(some_struct));
/* assign values to wanted fields*/
hashval = hash_bytes(&some_struct, sizeof(some_struct));
```

would result in hash_bytes() reading padding bytes as zero but the recent discussions of undefined behaviour have made me doubt this. (freebsd:6)
- I’m not super-confident that it will work, but my understanding of the C abstract state machine leads me to believe that it should work. An object has to be accessible via a character array, and writes to member variables should act only on the types in question, even if they are narrower than a natural machine word. (freebsd:8)
- I don’t know of cases where it padding bytes will be overwritten, but I wouldn’t be surprised if it happened. (gcc:0)
- Even though this is an issue, the production code I’ve seen made very sure to explicitly lay out shared memory to avoid overlap in boundaries. (google:1)
- This won’t happen if the struct was zeroed with memset or allocated with calloc, but likely if "zeroed" by just setting all members to zero individually. (google:5)
- I’ve seen code that prepared structs to send over the network. Some of these structs had padding. (google:14)
- I don’t know that the standard guarantees how padding is treated, and as it is often less efficient to write smaller values, I would not be surprised if there are cases where the compiler will write an entire word over a padded byte. (google:21)
- For hash comparisons, it’s more efficient to create the hash of all the bytes including the padding. (google:23)
- I’ve not observed any compiler writing junk, nor optimizing away reads, but it wouldn’t surprise me if one or more compilers did those things. (google:32)
- I think in practice you can do this in many cases (using static structs, or calloc, or memset), and where the compiler’s optimizer isn’t doing something particularly clever, but that you shouldn’t depend on it working because the value of padding bytes isn’t guaranteed to be anything specific, even if you just wrote them, and certainly not after a struct assignment or initialization of a local variable which may not even have written them. (google:34)
Although on the platforms I am on I don’t see any problems with this, platforms exist that don’t support smaller than word-sized writes. This can cause junk to be written due to some IMHO valid compiler optimizations.

Example code where I could expect this:

```c
struct {
    int8_t x;
    int32_t y;
} s;

void w(int32_t x) {
    s.x = x;
}
```

On platforms with only 32-bit write instruction, the compiler has two choices:

- Read 4 bytes at &s.x, change the first 8 bits to x, write 4 bytes at &s.x
- Write x at &s.x

This would leak the upper 24 bits of x.

Most platforms however have type-appropriate write instructions, and I wouldn’t expect compilers to overwrite the padding there.

Why would code rely on this? For security (not leaking data). There is virtually no way to prevent such leaks from C given the content of these bytes is unspecified, so virtually any code that reads/writes whole structs can leak unwanted data to disk. Only workaround is to make sure no padding happens, or not writing whole structs. To find such code... should be an easy search: http://sources.debian.net/src/systemd/215-18/src/journal/catalog.c/?hl=359#L359 is something I could quickly come up with. This at 64-bit alignment could break. (google:37)

- I’m fairly sure it would not violate the spec for this to not work, but in the simplest implementation it should, and I haven’t observed a compiler behaving otherwise. (google:38)
- I’d also avoid doing this, as it might defeat some compiler optimizations, e.g., promoting struct members to registers. (google:40)
- I’d *think* (gut feeling) that this is guaranteed by the standard. I know that it’s different for bit fields, though (bits of the word that are not declared with a name can be arbitrarily overwritten).

(I don’t quite get the "compilers would optimize away those reads" argument. The compiler should just be able to output reads and writes at the appropriate bit length that doesn’t touch the padding at all (unless your architecture is crazy and only allows accesses at word size or something).) (google:43)

- only sometimes (when?): I’m guessing that things like this

```c
struct { char x; short y; } s; s.x = 0; s.y = 0;
```

might well be optimized to a single 4-byte write, which will overwrite the padding between x and y. (google:45)

- Compilers are free to assume code doesn’t invoke UB, reading padding is UB. Compilers may optimize away those reads. (google:46)
- I haven’t observed the two behaviours above because I try not to rely on that, but they’re both legal (google:47)
- Just use memset, no? (libc:1)
- never observed, but i think it is a reasonable optimization to modify padding bytes around bit fields:

```c
struct {
    int x : 20;
} s;
s.x = -1; // set all bits to 1 including padding ones
```

i think in practice gcc always writes original content back to padding bits.
i consider relying on padding bytes behaviour to be a bug, but i think there might be corner cases where it is not easy to catch:

```c
union {
    struct { int i; char x; } a;  // ends in padding
    struct { int i; char x; char y; } b;
} u = {.b = {0}};

u.a.x = -1;
```

```c
u.b.y; // expected to be 0
```

(accessing u.a.x when the union "currently contains" b is valid because it’s in the "common initial sequence", the standard only allows read access though).

(libc:3)

- Presumably "zero all bytes" means memset(&s, 0, sizeof(s)).

One of many examples: zeroing structures before writing them to disk or across a network. (libc:6)

- I’m not sure what other compilers do, but this should work fine in LLVM. (llvm:2)

  - When you say zero all bytes I take it you mean copying zero to a ptr to the struct across a width of sizeof bytes. By padding I guess you mean bytes that are NOT covered by members of the struct. If that is what padding is, we do not know if anything is there or not. The member vars are reliable. I suppose you could use the padding to carry nefarious information. It would be safe from changes in our shop. (llvm:3)

- I’m not sure I quite understand this question. By padding, I assume you are talking about padding added between fields which would not be visible to the programmer; so, I’m not sure what your third question would be asking. Unless, one is talking about unions or other aliasing. In that case, one had better stick to accesses using the same union (or other overlay) for the storage’s lifetime (or the results will not be portable).

One case that is important is zeroing a structure before filling in the fields when one is going to treat the entire structure as a string of bytes for hashing, comparing or doing I/O. If one can’t ensure the padding gets (and stays) zero, these algorithms will not have predictable behavior. (llvm:5)

- Zeroing struct memory (e.g. with calloc or memset) should make reads from that memory predictable; Regardless of which struct members are written, I would expect the compiler not to write "hidden" data into that memory region.

Zeroing memory using struct literals (see example below) may or may not use memset; I would not bank on it to clear padding bytes.

```c
struct Point {
    int x;
    int y;
    char z; /* Padding in bytes 9, 10, and 11 */
};
```

```c
struct Point p = { 0 }; /* memset, or not? */ (llvm:7)
```

- I don’t know of any compilers that would do this (hence the "yes" for part 1), but I think a compiler could be within its rights as per the language standard to (for example) shorten the length of a memset() operation to not cover padding bytes at the end of a struct, leaving them uninitialized. (And it wouldn’t surprise me enormously if, say, a future release of GCC started doing this.) (regehr-blog:17)

- On x86, stores of different sizes require instruction encodings of different lengths. I wouldn’t be surprised by a compiler that takes advantage of a short encoding that writes into some padding bytes, but I haven’t noticed a compiler doing so. (regehr-blog:19)

- Seems like undefined behaviour, possibly a bus error (in practice) on some errors too (regehr-blog:22)

- In my answers (also the following), I write what I think a normal C compiler should do. I know that any piece of C code (other than Pascal transliterations) with more than three tokens likely contains undefined behaviour and the current
t bunch of C compiler maintainers therefore feel free to compile it to anything they want (except if it is a benchmark). (regehr-blog:28)
- X11 depends on this to not leak data over the wire protocol, but X11 generally declares padding bytes as explicit structure members instead of relying on a given architecture’s padding rules. (x11:0)
- I had a long time bug filed with gcc about structure copies and bitfield only copying the defined bits.

```c
a = b; // structure copy.
```
memcmp(&a, &b, sizeof(a)) could return non-zero since the memory occupied by a did not get filled entirely by b. (x11:3)
- Compilers will assume that padding bytes don’t need to be initialized and optimize away part or all of the zeroing.

So it will (probably) work as long as the zeroing happens somewhere the compiler can’t see the type, such as in custom allocation functions that return zeroed memory, when the virtual memory system zeroes fresh pages, etc. Calls to bzero/memset (and probably calloc) are not safe in this regard.

That said, while I haven’t seen a compiler write trash to padding bytes, it wouldn’t surprise me if it happened.

(Also, leftover unused bits in words used for bitfields, which are not technically padding bytes IIRC, are much more likely to accumulate trash than alignment padding bytes as they’re expected to be written to; whereas storing values into an alignment padding is wasted computation and relatively unlikely to appear gratuitously.) (x11:5)
- We rely on memset() or initialisation "foo = { 0 }"; to prevent stack contents being leaked into guests.

Inside Xen, the structures are filled using field names, so no explicit access to padding bytes. Copying data to/from userspace is done with hand crafted assembly, so will be unaffected by compiler undefined behaviour. (xen:3)

[2/15] Uninitialised values

Is reading an uninitialised variable or struct member (with a current mainstream compiler):
(This might either be due to a bug or be intentional, e.g. when copying a partially initialised struct, or to output, hash, or set some bits of a value that may have been partially initialised.)

undefined behaviour (meaning that the compiler is free to arbitrarily miscompile the program, with or without a warning) : 139 (43%)
[*] going to make the result of any expression involving that value unpredictable : 42 (13%)
[*] going to give an arbitrary and unstable value (maybe with a different value if you read again) : 21 (6%)
[*] going to give an arbitrary but stable value (with the same value if you read again) : 112 (35%)
  don’t know : 3 (0%)
  I don’t know what the question is asking : 2 (0%)
  no response : 4

If you clicked any of the starred options, do you know of real code that relies on it (as opposed to the looser options above the one you clicked)?

yes : 27 (11%)
yes, but it shouldn’t : 52 (22%)
no, but there might well be : 63 (27%)
no, that would be crazy : 80 (34%)
don’t know : 10 (4%)
no response : 91
Comment
- uninitialized variables contain whatever the memory contained at the moment
  of creation. But that memory is stable unless the variable is written to. (main2
  .2:3)
- a program may randomly initialize a random seed by not initializing it :-) (main2.2:5)
- Once I saw (and fixed) a much more complex variant of a piece of code as follows. I do not remember where the code was from.

```c
int f(int condition)
{
    int x;
    int a = 1;

    if(condition) {
        x = 42;
    } else {
        a = 0;
    }

    return a * x;  // x is uninitialized iff a == 0
}
```

- Several situations (multi-core communications software, multi-thread, etc),
where a variable or struct member is initialized by code other than the running
one. Core startup code looking at value of variables prior to the last reset, o
r synchronizing prior to clearing BSS. Among other examples. (main2.2:8)
- I think that compiler writers have implemented the option I have clicked (or
would like to).

But in reality I would expect a sane compiler to a) give an error/warning and b) actually do one of the starred options. Having a single uninitialized read some
where in the program potentially cause something completely unrelated to fail i
s not something that it is possible to debug. (main2.2:9)
- Know of Debian SSH key bug that was caused by this. So no "current" real cod
e. (main2.2:15)
- Depending on what storage the object is in, the uninit fields might have dif
ferent values each time you execute the program. (main2.2:18)
- it depends on whether the type has trap values: if so it really is undefined
behavior; also if the variable might be in a register, a special clause from th
standard makes it sometimes undefined behavior (main2:0)
- On an embedded system, I expect a read of the given address hell or high wat
er. Stability would relate to the contents accesses at that address. (main2:6)
- Technically UB, but afak compilers just treat the read as a poison value. (m
ain2:7)
- I think the standard calls it "undefined", but I haven’t seen a compiler tha
t would miscompile this. The compiler may optimize away the read. I see value in
e.g. "a=undefined; a^a" evaluating to 0, but I’m not sure that compiler writes
find that useful. (main2:8)
- Again no benefit, so play it safe. (main2:9)
- I’ve fixed a lot of code like this before. (main2:10)
- is also sometimes intentional to get entropy for random number generators.
(main2:11)
- I guess it really depends on where the value is defined. Global variables co
uld be expected to be 0 (main2:14)
- Definitely undefined, though mistaken "security" code has done this. (main2:
20)
- I would expect, in practice, that an uninitialized struct member yields an a
rbitrary but stable value. I haven’t tested this. (main2:27)
- There’s much code that depends on the value being stable, usually implicitly,
to operate predictably. But such code is buggy.

Some uninitialized variables access by the standard are undefined, others are un
predictable. Depends on if you take its address. But even having just checked the standard (cheating I know), it is unclear to me what the right answer here is.

- Only code I know of which used uninitialized values was OS code, reading memory below the current stack pointer to dump exception stack unwind details after the unwind had happened. Interrupts were disabled at the time to avoid the stack space being reused while dump happened. (main2:29)
- In practice the answer is likely to depend upon how the memory is allocated. e.g. Static variables may be initialised directly from the object code - undefined in theory but repeatable in practice. Structs that are ‘malloc()’ed may be undefined and non-repeatable. Some compilers play safe and use zeros to fill uninitialized memory. (main2:32)
- There are actually many questions combined in one here. Also I am aware of a significant divergence between the LLVM community and MSVC here; in general LLVM uses "undefined behaviour" to mean "we can miscompile the program and get better benchmarks", whereas MSVC regards "undefined behaviour" as "we might have a security vulnerability so this is a compile error / build break".

First, there is reading an uninitialized variable (i.e. something which does not necessarily have a memory location); that should always be a compile error. Period.

Second, there is reading a partially initialised struct (i.e. reading some memory whose contents are only partly defined). That should give a compile error/warning or static analysis warning if detectable. If not detectable it should give the actual contents of the memory (be stable).

I am strongly with the MSVC folks on this one - if the compiler can tell at compile time that anything is undefined then it should error out. Security problems are a real problem for the whole industry and should not be included deliberately by compilers. Personally I think the LLVM folks who are detecting undefined behaviour and deliberately generating bogus code, should be thrown out of the ACM and IEEE on the grounds of having violated the ethics clause. (main2:37)
- I think some of the options are not mutually exclusive. If it’s "going to give an arbitrary" value then surely it’s also the case that it’s "going to make the result of any expression involving that value unpredictable". (main2:46)
- In the old KAI C++ optimizer (which worked for C too), "if(uninitializedvar) foo(); bar();" could execute *both* foo() and bar(), or maybe neither. (main2:49)
- I would like to hope that uninitialized data is a niche if not non-existent problem these days, given that every compiler worth its salt has solid warnings about such things, on by default. But I don’t trust people. (main2:51)
- My understanding (which could be wrong?) is that it’s not all that uncommon to read, say, uninitialized ints and e.g. add or xor them -- and only later do other program state checks and decide that those values are not needed -- but that it’s basically safe to do that as long as you’re not using uninitialized values (or results of computations involving such values) for e.g. branches or array indices. (main2:54)
- Any code that memcpy’s potentially uninitialized or partially-initialized storage relies on this. LLVM is moving towards treating this as UB in the cases where the standards allow it to do so. (main2:69)
- Without optimizations: Though likely technically "undefined behavior", a declared variable, or especially a struct member, should already have space allocated, and simply result in reading whatever data was last left in that location. With optimization: Wouldn’t be too surprised to see this situation result in undefined behavior. (main2:72)
- I know of lots of code that deliberately copies uninitialized values around: member = other.member;
and the fix is to swap that out for memcpy.

The other case people talk about is using uninitialized values as a source of entropy. Those people would be crazy.

Instead of full fire-breathing demons, our compiler will only ruin (make unstable...
e) the expressions that depend on the uninitialized value. (main2:74)
   - Reading uninitialized values is one of the most famous undefined behavior. (main2:80)
   - Often values will be chosen, sometimes special token values to signal that this is happening. But strictly speaking, it's undefined, so it is *allowed* to do anything, crashing, warning, sticking in some value, whatever. (main2:81)
   - These answers apply to global variables. Nevertheless global variables may or may not be initialized depending on variable placement/compiler options (IAR compiler allows different memory segments where initialization takes place and others where not). (main2:82)
   - I saw code using this to increase entropy to a PRNG (though required a special mode to avoid it when testing with valgrind) (main2:91)
   - Found a few of these in real "optimized" code with clang static analyzer (main2:98)
   - AIUI there are complicated restrictions on the compiler’s freedom to miscompile here, but trying to navigate them seems like a rather bad idea.

That said miscompiling a memcpy or assignment of a struct object which happens to have one member uninitialized seems rather harsh.

My preferred behavior would be that the language guarantee initialization of automatic variables in the same way they do statics. (main2:100)
   - I'm aware of a bug in the seeding of a random number generator that was caused by including uninitialized variables in the seed which caused the whole value to be poisoned and 'optimized' out leading to deterministic seed values. So there definitely _was_ code that relied on it, but that particular bug has been fixed a long time ago. (main2:108)
   - I don't think I've met a compiler yet that actually miscompiles in this case, but the result is still a broken program, and you usually get a warning. Your coding standards insist on compilation without warnings, right? (main2:111)
   - Might as well be "undefined behaviour", I don't know the standard so well, but this is what I observed. (main2:117)
   - zlib does this as an optimization (read and operate on multiple bytes at once from a buffer). The read value doesn't actually affect the output, so it's safe. (main2:129)
   - IIRC bzip2 had some conditional depending on an uninitialized value (irrelevant in that case), later fixed due to valgrind complaints. (main2:137)
   - This is well defined behavior in many cases, e.g. if the variable or struct overlays memory-mapped registers. (main2:138)
   - Sometimes used as cheap insecure source of random bits (before the era of secure data where it’s common for the operating system to erase deallocated pages). Often, uninitialized data is "don't care" values which are assumed to be safe due to overwriting or unused. Compilers used to be poor at tracing conditional paths (pages of mostly spurious warnings got ignored), or code depends on dynamic data taking a consistent branch. Unsafe uninitialized data may have only a 1-in-million chance to cause a visible error - it could be ubiquitous. (main2:147)
   - the compiler shouldn't be making this choice; the data should be created at runtime with whatever junk is there. it would be stable, but unreliable as it would change for every memory chunk. (main2:148)
   - running anything under valgrind for the first time, you’ll get a UMR. bad scene. (main2:156)
   - You shouldn’t see "unstable" values with a data structure changing in memory between multiple reads, unless it was declared volatile and there is another task that might write to it. Reading an uninitialized variable should not allow the compiler to mis-compile the program, but the values are unpredictable (for example, automatic variables that are not initialized when a function is entered will have values that may be left over from a previous use of memory). (main2:158)
   - I worked on a code base that used uninitialised memory as the seed to its random number generator.

Also, in the old days game cartridge save RAM was memory mapped. The only way to know whether there was a save was to look for a magic number written out to the start of save RAM.

Also, I doubt the compiler has enough information to know what is uninitialised (in which case WARN US!!) so the current behaviour is good because it’s predict
able. (main2:163)
- Notoriously, Debian’s OpenSSL implementation relied on this to generate a random seed (2008). (main2:165)
- answer is what Gil+Viktor think LLVM will do, based on some knowledge of its internals. They don’t know why the LLVM people think it’s useful to give stronger guarantees eg for AND with a constant. (main2:172)
- Some code may assume that they are initialized to 0.
I think the compiler should either read the actual value in memory, or emit a clear warning if it is optimizes more (behavior preferably controllable by a flag).
- (main2:173)
- some encryption tool uses uninitialized values for "more" entropy in its RNG.

llvm actually will set the value to "undefined" which propagates. (main2:176)
- I did discover an interesting issue related to this one. In clang, if you declare a function variable const, but fail to initialize it, it will be put in a section and no warning or error will be raised. IMO such a variable should be an error.

Sample:

```c
void fun()
{
    const char foo[16];
    /* do something w/ foo */
} (freebsd:5)
```

- I recall bumping into code that assumed that the stack was initialised to zero (I don’t recall where, unfortunately) and broke when the code was dynamically linked because the dynamic linker was dirtying the stack.

I don’t believe that reading an uninitialised variable will lead to the compiler arbitrarily miscompiling the program but I may be wrong. (freebsd:6)
- I only know of some obscure self-encoding code that used this to check known stack values to crash on debugger or other interrupts. (google:4)
- undefined behaviour might include all of the starred options (I have seen all three of them, but "arbitrary but stable" seems the most common one). (google:5)
- I don’t mind if an optimizing compiler makes its own life easier by picking a value for the uninitialized value, that generates less code. But that value needs to be consistent. (google:7)
- I’ve seen two different ways that code makes assumptions about uninitialized data. One is assuming that it’s initialized to 0 (often this is a C++ programmer incorrectly assuming default construction) and using it in a boolean context. The other is storing a copy of the data without regards for whether it’s unitialized or initialized and then subsequently comparing the value to the earlier copy to check for modification. The former is definitely a bug. The latter depends on the value being stable. (google:11)
- Note that the first answer is probably correct, but it is meaningless to talk about "undefined behaviour" "with a current mainstream compiler" -- either you’re asking about the standard or you’re asking what a real compiler will do there. (google:14)
- I would have said "no, that would be crazy", but I’ve seen enough crazy code already to be that optimistic (google:17)
- I believe OpenSSL will sometimes read uninitialized variables to provide input for its random number generator. (google:18)
- Most compilers warn on this, now, in my experience. (google:21)
- This is technically undefined behavior, but I’ve never encountered a case where it did arbitrary miscompilation.

In real world usage, I’ve seen random number generators set the seed values using XOR with uninitialised memory instead of =, as an extra source of memory. (google:29)
- Leaking or using "random" data from uninitialized variables is a common security issue - especially as said data may be attacker controllable due to reuse of freed memory blocks. Don’t know any concrete example though, but I bet the CVE
s shouldn't be hard to find. (google:37)
- Normally, this will give you an arbitrary but stable value, but I believe that per the spec it’s undefined. (google:38)
- Debian, let’s not try to get entropy from uninitialized memory when generating keying material kthxbye. (google:39)
- It usually leads to a crash. (google:41)
- I think this might be undefined behavior per the standard but no sane compiler should miscompile it. I’ve seen plenty of code access uninitialized values (and throwing the result away or something) and that should be fine. (You should probably not rely on it being stable, though, although I can’t imagine why it shouldn’t be.) (google:43)
- openssl used to copy uninitialized data into its entropy pool (not sure if they still have that bug).

zlib uses nasty optimizations in its longest_match function that reads uninitialized data and then conditional jumps based on it (however in the end the match length is limited by the available bytes so "it’s ok")

python memory allocator used to do it (reading memory that may be uninitialized, but later checked if that was the case)

boehm gc is probably an example for most of the non-standard assumptions (eg. it scans the stack for pointer references).

(i’m sure there are many cases where it is done by mistake) (libc:3)
- I’d say "undefined behaviour", but I disagree with the characterization of "arbitrarily miscompile". I don’t hold with the old Fortran adage of allowing missile launch.

I think of it more as an extended form of constant propagation. Since the value is arbitrary, the compiler can choose a value for the uninitialized value that will allow it to fold a larger expression. Including a value that allows a loop to never be entered, and thus eliminated.

As for real-word code, I can imagine that there are programs that rely on uninitialized values not starting WWIII, and eliminate the results of any expression using uninitialized values via alternate logic. (libc:6)
- I believe OpenSSL has some code that does this. Apparently they think it gives them some extra randomness. But it also makes valgrind complain about apps which use OpenSSL, <http://www.hardening-consulting.com/en/posts/20140512openssl-and-valgrind.html>. Although I’m not sure if they do it in C, or only in assembler. (libc:7)
- Simply copying uninitialized bytes does not in practice cause problems, and we try hard to make this work in LLVM. It also allows us to speculate loads that might be uninitialized, which is important.

Memory Sanitizer also takes great pains avoid warning on copies of uninitialized data, and instead flags "uses" of uninitialized data, like conditional branches and passing it to external libraries. (llvm:2)
- We see this occur and we correct it once discovered. Uninitialized variables make debugging difficult, it is not desired. (llvm:3)
- I’ve seen code that worked for decades with defects like this Until it finally broke :-). I consider these defects and fix them when I find them. (llvm:5)
- Reading and using the contents of an indeterminate lvalue could be argued not to be UB for some versions of the standard in some circumstances, but based on compiler behavior, you might as well assume that reading indeterminate values is UB. http://blog.frama-c.com/index.php?post/2013/03/13/indeterminate-undefined

Reading the contents of an lvalue for the purpose of copying to another lvalue without any computation (not even a conversion) is the useful action that the C99 standard should have chosen as allowed by exception. Unfortunately, it didn’t. Nevertheless, this is allowed in Frama-C’s value analysis and CompCert’s semantics, because structs with padding need to be copied (either with lvalue = lvalue assignments or through memcpy()). (regehr-blog:0)
- It’s UB but feel that current compilers tend to go with the "arbitrary and u
- I think "relies" is a misnomer, but accurate in the case of bool values. Since 0 is false, and nonzero is true, it’s easy to get the "right behavior" in a consistent enough manner for uninitialized variables that rely on true being the correct default value. (regehr-blog:9)
- Can’t give an example, but I’ve seen plenty of code that assumes uninitialized variables will be zero, presumably by accident rather than naivety. (regehr-blog:10)
- I think it is either an arbitrary, unstable value, or undefined behavior for _Boolean. (Certainly the C++ bool type can behave very strangely on at least some compilers when uninitialized.)

(I think the C11 spec allows implementations to define specific bit patterns as "trap values," which result in undefined behavior when read, but outside of _Boolean I don’t think any real-world compilers actually do this. I know this because I had to look it up recently.) (regehr-blog:12)
- I assume we’re not talking about static variables — which are never uninitia-
lized. (regehr-blog:14)
- I’m assuming that we’re not using the volatile modifier (regehr-blog:29)
- I believe some code, such as OpenSSL, used to rely on it for random data, but has since been fixed. (xen:2)
- I fail to see how either of the upper two starred options is remotely safe — in order to get an unstable or unpredictable result the compiler must "know" th
at the read isn’t an ordinary read of an ordinary variable slot, in which case a
rbitrarily wrong other behavior can easily ensue.

I would expect as a QoI thing that any situation where the compiler is aware of a read of an uninitialized variable would result in a warning; but I’m also aware that this is naively optimistic, and that there are situations involving loops where the sensible behavior to expect of a sane compiler is not at all clear. (xen:5)

- I remember a bug related to Debian’s SSH key generation where Valgrind found reads to uninitialized memory and the fix consisted of memset()ing them to 0, w
hich in turn spoiled the RNG which used them for seeding and led to a very low n
umber of identical SSH keys being around. Must have been around 2008. (xen:1)
- Not code that *intentionally* relies on it (hence the "shouldn’t") but some-
times programmers make mistakes, and the code works because it just happens to ge
t a sensible stable value. (xen:2)
- I tend to find that compilers which can spot unitialised variables will comp-
lain about them. It is the times where UB can’t be spotted where bugs start to
bite. At this point, the compiler has done its best to compile the code correct-
ly. (xen:3)

-----------------------------------------------
[3/15] Can one use pointer arithmetic between separately allocated C objects?
If you calculate an offset between two separately allocated C memory objects (e.
g. malloc’d regions or global or local variables) by pointer subtraction, can yo
u make a usable pointer to the second by adding the offset to the address of the
first?

Will that work in normal C compilers?

<table>
<thead>
<tr>
<th>Choice</th>
<th>Yes</th>
<th>Only Sometimes</th>
<th>No</th>
<th>Don’t Know</th>
<th>I Don’t Know What The Question Is Asking</th>
<th>No Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Votes</td>
<td>154 (48%)</td>
<td>83 (26%)</td>
<td>42 (13%)</td>
<td>36 (11%)</td>
<td>3 (0%)</td>
<td>5</td>
</tr>
</tbody>
</table>

Do you know of real code that relies on it?

<table>
<thead>
<tr>
<th>Choice</th>
<th>Yes</th>
<th>Yes, But It Shouldn’t</th>
<th>No, But There Might Well Be</th>
<th>No, That Would Be Crazy</th>
<th>Don’t Know</th>
<th>No Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Votes</td>
<td>61 (19%)</td>
<td>53 (16%)</td>
<td>99 (31%)</td>
<td>73 (23%)</td>
<td>27 (8%)</td>
<td>10</td>
</tr>
</tbody>
</table>
If it won’t always work, is that because [check all that apply]:

- you know compilers that optimise based on the assumption that that is undefined behaviour: 51  (100%)
- no response: 228
- other: 51
  - it’s not standard.
  - It is technically UB, but I’ve never actually tried it.
  - see below
  - It would completely confuse alias analysis I believe.
  - I believe that CHERI credential-pointers don’t support this
  - It relies on luck
  - systems with odd memory layout (e.g. embedded) won’t work
  - Standard says it is undefined...
  - May depend on architecture
  - you know compilers that optimise based on the assumption that that is undefined behaviour, not reasonable to use ptr arith outside single assignments
  - This seems crazy. Even in your local mapped memory, something like Malloc is unlikely to give contiguous memory blocks for each reservation
  - C permits a segmented address space
  - architectures with segmented memory. E.g. 8086.
  - segmented memory models
  - you know compilers that optimise based on the assumption that that is undefined behaviour, segmented architectures
  - The difference in addresses may not be divisible by the structure size. But that problem can be avoided by casting to char*.
  - This definitely won’t work on Harvard architecture micros with different instruction/data access instructions.
    - as/400.
    - Weird segmented memory architectures might not like it
    - Data type sizes may vary
    - the pointers may not be diffable.
    - page mapping may change.
  - There is no guarantee about the order of allocation in the heap (could be backward if it pleases the implementer of malloc).
  - NUMA, multiple allocators
  - Separate address spaces
  - compilers may start to optimize based on that, if they don’t do that already
  - I have to assume that a compiler may optimize based on the assumption that this leads to undefined behavior.
  - alignment issues
  - alias analysis may enable non-behavior-preserving optimization of memory access
    - On segmented architectures with far/near pointers the subtraction will not work.
  - we use memory sanitizers and address sanitizers that cause termination of this sort of thing is occurring
  - I don’t know of compilers that optimize based on this, but I wouldn’t be surprised if it existed
    - see below
    - objects may exist in different address spaces
    - different underlying system memory models
    - I can’t imagine a compiler writer who would make sure this behavior is supported.
  - no-typical memory models
  - depends on the pointer type
  - breaks language semantics
  - Segmented addressing
  - It depends on the type of the objects I guess?
  - Segmented memory architectures
  - Some platforms have strange memory layouts.
    - this assumes a "flat" (non-segmented) memory model to start with.
  - possible security features of some c libraries?
  - Segmented architectures
Comment
− pointer arithmetic is arithmetic; after subtraction and further addition the result will be the same, i.e. a valid pointer. (main2.2:3)
− because when allocated separately, extra padding is added before the actual object for the heap manager, plus you have no control over where in the heap the objects are allocated, there is a big chance that they’re not consecutive. (main2.2:5)
− Real programs expect to be running in a flat address space (or if they aren’t they are coded to know about the lumpy bits).

If I was writing code that needed this I would expect it to work, though as far as I remember it isn’t actually legal according to the standard and pointer arithmetic only works inside objects. (main2.2:9)
− Memory allocators rely on it. (main2.2:17)
− It’ll be fine until a malloc’d object gets realloc’d or free’d, or the function with the local returns. (main2.2:18)
− Pointers might live in different address spaces (main2:0)
− I haven’t come across a compiler that uses this as optimization opportunity. As soon as a benchmark benefits from this, compilers will implement this optimizations, forcing people to manually cast to an integer. (main2:8)
− I’d distinguish casting pointers into the same allocated region into differently typed objects, and forming pointers between those, which I do do, from separately allocated objects, which I don’t. (main2:9)
− This is mostly a problem with hardware architecture like GPUs. (main2:10)
− I’m certain such code exists.

It won’t work the the CHERI compiler as object boundaries are hardware enforced (main2:16)
− If the pointers are cast to intptr_t first and the arithmetic is done there, then this is guaranteed to work. (main2:18)
− I know this isn’t valid according to the standard but it seems unlikely that a compiler writer would gain anything by not doing "normal" math on the register values that represent the pointers. (main2:20)
− I think it will work, but it is still undefined behaviour. (main2:22)
− I would expect this to be safe in practice today, but not safe from future improvements in compilers and optimizers. (main2:27)
− Code that depends on this is highly compiler dependent, but things like old-school alloc used to use this trick to see which way the stack grew and do evil things based on that. (main2:29)
− Marshalling code is sometimes written by computing offsets within a struct (eg by using the offsetof() macro) and then using those offsets together with a base pointer to read or write fields of a struct when marshalling it.

This is pointer arithmetic between two separately allocated objects: NULL as part of the offsetof() macro implementation, and the base pointer when using the offset. (main2:30)
− This relies upon an assumption that you may subtract unrelated pointers. Pointers could be implemented as a base address plus an offset.

In theory the following compiler option would help improve code quality:

Implement malloc() so that it allocates an additional word before the memory requested. Store the length of the memory in that word. Implement pointers as the tuple (base, offset). When using a pointer check that (offset >= 0) && (offset < length). The optimiser may remove redundant checks where it is clear that the offset always falls within this range. (main2:32)
− The prohibition against this (and some other wacky restrictions on pointers) in the standard was only put in there because at the time of the standard the 8086 segmented memory model had not yet died out.

Most of that text in the standard reads as "sensible thing x except that you shouldn’t really do y because if you are on 8086 in segmented mode it won’t work". (main2:37)
− The MPI Forum (which includes me) recognizes the problems of address arithmetic in C and has utility functions to make it possible to do things that are necessary, but in a portable way (of course, the implementation is platform specific)
− My real answer to the first question is "I’d hope so, but I fear it may not"
− I believe the spec says it is undefined and you shouldn’t do it. (main2:55)
− For code that uses arena memory allocation, pointer math might be important, between objects allocated within that arena. It’s never been clear to me where to draw the boundary between defined and undefined behavior in such a case. It’s also never been clear to me, how optimizations based on this part of the spec would actually help. (main2:59)
− I’ve seen the code that was doing something around the line of
void push_back(const T& v) {
    if (&v >= begin() && &v < end()) // we need to preserve the "v". (main2:64)
− It’s undefined behavior, but an implementation is permitted to use undefined behavior in its own code since it ostensibly has control over it. An example of this is the glibc strcpy source (generic C version) using a ptrdiff_t between src and dest to create a single offset and then walking through only one pointer. (main2:70)
− Firstly, what do you mean by "allocated"? A global variable may have an address that is chosen by the programmer (consider a linker script that places globals at known addresses).

In any event, if I see you walking outside the range of a malloc’d pointer then I’m happy to give you undefined behaviour. In practice, we have to make so many conservative assumptions that we don’t manage to do it much (ever?).

The most austere interpretation is that the runtime environment may create pointers by allocating an otherwise-meaningless entry in a hash map (ie., ptr+1 allocates a new entry in the hash_map with no arithmetic relationship to ptr -- implementation of < etc. left to the reader) and that precludes pointer accesses. (main2:74)
− Illegal by the standard (main2:79)
− Most commonly I see this with offsetof used to assign value to structs members by their offset, which I think should still be considered wrong.

I don’t actually know of compilers that optimize this away, but perhaps there are some. If GCC has a flag for it, then I do not know about that flag doing this. (main2:81)
− It is hard to tell what ‘separate allocations’ means given the loose bounds set up by the introductory paragraph. It is very common to write custom memory allocators. Would the whole memory chunk, containing multiple separate objects, obtained by such allocator from malloc(), count as one allocation? Would each separate call to a custom allocator count as a separate allocation? How would the compiler know? (main2:90)
− QEMU relies heavily on pointer arithmetic working in the "obvious" way on the set of machines/OSes we target. I know this isn’t strictly standards compliant but it would break so much real code to enforce it that I trust that gcc/clang won’t do something dumb here. (IIRC there was a research project that tried to enforce no buffer overruns by being strict to the standards text here and they found that an enormous amount of real world code did not work under their setup.) (main2:93)
− This is undefined behaviour.

Will probably work on POSIX; will not work on segmented architectures.
Allocators and linkers need to do things like this.

Best to convert to uintptr_t before playing tricks like this. (main2:94)
− The XOR linked list, as described by Wikipedia, does pretty much that. It’s subtraction instead of xor, but it’s pretty much the same anyways. (main2:96)
− In practice I don’t know what mischief any given compiler might get up to.

While programs aren’t formally allowed to rely on it, system software (e.g. a malloc implementation) may have to do it. It would be rather upsetting to find a C compiler that couldn’t compile libc! (main2:100)
− I’ve seen this done in an OS to link system function calls into ELF binaries (main2:112)
Generally a compiler doesn’t have the ability to know whether two pointers come from the same memory region or not. The offsetof macro is sometimes implemented in terms of pointer arithmetic involving the null pointer. (main2:115)
- I think it will work in practice with most compilers with the default flags even though it is undefined. (main2:123)
- I think this is undefined behavior, but I’d be surprised if it didn’t always work. The compiler would need to know for certain that they’re separately allocated.

I’m unaware of code that relies on this specifically, but I’m aware of code that casts it to an intptr_t (or whatever) and does the comparison on that. (main2:126)
- Layout of objects pointed to by the same pointer type is commonly assumed to be identical. i.e. for each type it is assumed there is only one layout. (main2:127)
- Not true for every system triple. But typical systems offer uniform memory architecture. Different allocators might use different granularity (scale). Or nonmetric opaque handles. (main2:147)
- relative pointer offsets; this is often used in compression and, while not a good idea for disparate memory allocations, should function the same. also, what constitutes an ‘allocation’ is different between systems. if a heap is locally managed and fixed in size, all allocations can be disparate, yet offsets are constrained. (main2:148)
- I know the C spec has very subtle wording around addresses not necessarily being numbers; I also know of tons of code that very explicitly relies on them being numbers, maintained by teams who completely dismiss platforms and compilers which don’t support it. I’m hesitant to put "yes, but it shouldn’t" here because it’s so embedded in the culture that at this point it’s de facto behaviour compilers need to support on platforms where it’s the reasonable implementation (i.e. flat memory models). (main2:156)
- In C, "global" variables is a misnomer. There are variables of the static storage class, variables declared outside all scopes, and variables with external linkage. All three of those things have to be true for a variable to be what we think of as "global." There is not a single thing that makes a variable "global" so the term should not be used when talking about C programs.

My intuition says that doing pointer math between statically allocated variables in the same source file would generally be safe, and similarly doing pointer math between automatic variables in the same function would generally be safe. Pointer math between separate mallocs, I’m not entirely sure of. I am not aware of what the standard says about this but I believe operating systems which use memory pages, as opposed to many simpler embedded systems, may have problems with that. And certainly there could be problems with variables that are allocated with different memory models, like the infamous near and far pointers. In x86 terms I think they would have to have the same base register. (main2:158)
- If I wanted to do this, I’d cast the pointers to integers, do all the arithmetic on integers, and cast back to pointers only at the end. (I’d also try to do some web searches to make sure that my vague memories about casting between pointers and integers being allowed is correct.) (main2:160)
- Anything that serialises structure with pointers is going to do this.

Also, I haven’t worked on a game yet that calls malloc(). It’s always system calls for a large block and then in-house custom allocators. malloc() is way too slow for games.

I’ve also worked on 64-bit ports of 32-bit code that purposefully keep 32-bit pointer-like ints to keep their memory footprint low (with appropriate calls to tell the system exactly where we want our data).

Also, how is the compiler going to know which allocations are separate?! (main2:163)
- If you are working only with heap-allocated storage and restrict your code to use a known implementation of malloc, this seems to me not unreasonable. Like wise for globals, e.g., if your are restricted to ELF files and know that the globals will all wind up in the static segment. But, with local variables, it seems to me extremely unsafe, because compilers are free (and should be) to use reg
isters instead of memory locations at any point in the program, so the supposed address of a given variable could easily be rendered garbage. (main2:165)

- Why only sometimes? First: Gil+Viktor think they can make the LLVM alias analysis break this. Second: Gil changes his mind; he thinks it’ll work. Viktor doesn’t know enough about the actual alias analysis. Third: they wouldn’t be surprised if this confused the alias analysis. (main2:172)

- I know it’s against the standard, but it is a common implementation for doubly-linked lists (usually xor, not add) (main2:176)

- I expect it’ll work in many cases, but relying on it would be crazy. (main2:177)

- This is somewhat complicated by the aliasing rules. If it goes through a char pointer, then it is allowed (to a degree), but w/ modern compilers, doing such a thing is normally not allowed, unless the types match. (freebsd:5)

- I’m fairly confident this was outlawed in the first ANSI C standard. (freebsd:6)

- It is difficult to imagine a compiler that could actually prove this in any but trivial and uninteresting cases. "A quality implementation will emit a diagnostic." (freebsd:7)

- I believe that compilers are not yet taking advantage of the restrictions on valid pointers in the C standard, since there are not obvious ways to speed up the resulting code. (freebsd:8)

- Used for calculating a fingerprint of bytes in memory, for FIPS validation. The OpenSSL FIPS canister is one example. (gcc:3)

- I imagine a compiler would optimise assuming it doesn’t happen, though I’m not sure if a particular compiler does it (gcc:4)

- It doesn’t sound crazy, and I’d assume it would work with a "normal" compiler on a "normal" architecture, but I can’t think of a valid use case for doing that...

(google:5)

- "No", because of e.g. segmentation in MS-DOS. MS-DOS lives (unfortunately). (google:14)

- I believe this is undefined behavior but I can’t think of how a compiler would take advantage of that for any optimizations. (google:18)

- A novice mistake would be to subtract the pointers Type1* and Type2*; the compiler should catch this because I can’t think of a meaningful value that would be returned. The better route is storing as ptdiff_t, and casting both pointers to char*. Though honestly, this is a little crazy in and of itself; a lot of code relies on char being a single byte, while the standard to this day makes no guarantee of that, and only recently started offering int8_t in C++ (the type in C is a little older). Even then, you’re still coding to architecture, and you’re relying on the two pointers existing in the same virtual memory space. I can easily see kernels doing this sort of thing, since they can make such guarantees. (google:21)

- I’m not totally sure on this one. I’ve seen something similar used in binder.c in the linux kernel does this via user_buffer_offset. It’s stored in the ptdiff_t type, which I read to be meant for pointers allocated together in a single array, which may imply this question should be no, but I’m unsure. (google:26)

- (1) Difference between T* is not necessary a multiple of sizeof(T).

(2) There could be funny memory models like x86 far pointers, or different memory namespaces like Harvard architecture.

(3) If it's char* and von Neumann architecture with plain address space, this shouldn't work.

(google:30)

- It seems crazy to assume, unnecessarily, that all of the data are in a single, contiguously addressable area. (google:32)

- You can’t access the second object with a pointer to the first without causing undefined behavior. I suspect there’s no requirement for the subtraction to produce a meaningful value. Possible practical reasons for this to fail are segmented memory, typed pointers, and bounds-checking compilers. I could imagine compilers taking advantage of this to perform additional optimizations, but I don’t think I’ve run into that myself. (google:34)

- Most real-world compilers put all objects in the same address space, so pointer arithmetic between objects will work. Some exotic compilers (e.g. for embedded systems, or DOS) don’t, but they are rare these days. (google:36)

- Some platforms, e.g. MS-DOS in some memory models, have 32-bit pointers and
16-bit memory addresses. There such arithmetics will obviously break. (google:37)
- This won’t work on systems that use 16-bit pointers and a segmented memory model, for example, and there are other cases where the logical address space is not flat. On most processors it will work fine, though it may get you smacked by a teammate. (google:38)
- Pointer difference is only valid between pointers to the same allocated region. (google:39)
- This depends on struct size/alignment. malloc typically aligns on the largest word size a machine supports, but if a struct that is larger than that the distance between the pointers may not be an integer multiple of the struct size, so the subtract result will not really be representable as an offset_t. (google:42)
- This works on any sane architecture, and it should because certain systems code (e.g. kernel memory management) needs to rely on the compiler not doing anything insane with pointers. For example, coreboot contains a mechanism to relocate part of its data segment from one base address to another during execution. All accesses to globals in that segment go through a wrapper which after the migration uses arithmetic like this to find the new address (e.g. something like ‘return !migration_done ? addr : addr - old_base + new_base;’).
- It might well work if memory model is flat, although it’s UB formally. Generally it will work if you allocate those regions by some external means (and memory model is right) -- this is then implementation specific, as it uses resources obtained not the standard way.

System code relies on that, but typically it was it’s own allocation functions to begin with. (google:46)
- Lots of code relies on this, I’m afraid, though generally it’s to heap objects. (libc:1)
- since c99, objects live in an abstract space not in a flat memory space, allowing usage that assumes flat memory space (pointer subtraction across different objects) breaks the semantics of the language (it matters for optimizations, but i think it is mostly important for allowing memory safe implementations of c where a pointer representation knows about the underlying object it points to)

i’ve seen code that tried to determine stack growth direction by comparing the addresses of two local variables.

i’ve seen lots of unsafe pointer arithmetics that (temporarily) went outside the bounds of the object where the pointer points.

i don’t know if current compilers base optimization on this but i think they should:

```c
int i;
// ...
char s[4];
char *p = s+i;
```
tells the compiler to assume that i is 0,1,2,3 or 4. (libc:3)
- I know code that does this, but it’s a sign of very badly-designed code and a bad code smell, and I would be dubious of any code that did this except in very tightly-constrained environments where the programmer knows lots about the compiler behavior. (libc:4)
- This is essential to writing system software, including malloc itself.

Therefore any production compiler has to have an option to disable reasoning based on the ISO undefinedness of this. (libc:6)
- In practice, I believe this works in normal C compilers because LLVM’s alias analysis (and I assume others) considers both the base and the offset of a pointer arithmetic operation as possible sources of objects. Computing the differenc
e between two pointers produces a value "based on" the pointers to both original objects, so it is possible to recover pointers to both objects.

In theory, my understanding is that this is illegal, but compilers are unlikely to break it anytime soon. (llvm:2)
- This practice is usually a bad idea. But we use it often when operating on arrays. It is fast and reliable. "p++ = *q++;" (llvm:3)
- Technically, this will work, but I can’t see any benefit to doing it (except when writing the heap allocator, of course). One would normally use offsets to save space or make the structures relocatable -- but offsets for arbitrary heap objects would have to be the same size as the pointers (no savings) and heap object's aren’t relocatable.

If you are working in an OS kernel, there are often requirements to use pointer arithmetic for arbitrary memory locations and with knowledge of page boundaries. .. (llvm:5)
- It only works with great care taken to cast the pointers to an appropriately sized data type (e.g. size_t), and then casting the result back to the correct pointer type. This is of course incredibly reckless and should not be done by anyone. (llvm:7)
- I know it’s UB, but I don’t know if compilers use it. (regehr-blog:1)
- My team is developing an alias analysis based on the fact that this is UB (regehr-blog:3)
- (p2 == p1 + (p2 - p1)) reminds me of a pattern that I have used: (p1 = pibase + (p2 - p2base)) when realloc’ing a buffer with a reference into it, but this doesn’t technically perform pointer arithmetic across allocs. (regehr-blog:7)
- On PICs and MCS51s, the two objects could actually be in different data spaces (e.g. RAM vs flash memory). It would be nonsense to do pointer arithmetic on them. (regehr-blog:8)
- I’d guess if you cast it to void* it will work? (regehr-blog:10)
- I’m pretty sure this is OK if you perform the arithmetic on uintptr_t, then cast back to a pointer type. The only use I can think of is an xor linked list (which could also be implemented with subtraction instead of xor), but I don’t know of any real situation where an xor linked list is the best data structure. (regehr-blog:12)
- Seems like an operating system written in C could well want to do this. (regehr-blog:14)
- The embedded systems I work on usually use malloc very rarely, but I wonder how this applies to statically allocated objects. (regehr-blog:15)
- I have seen cases where a C compiler for an embedded processor dealt with pointers to distinct address spaces for certain things (where this definitely wouldn’t work), but in the case of a "normal" hosted implementation I’d certainly expect it to. (regehr-blog:17)
- In flat address space it is expectable behaviour, because pointer is just an single integer index.

That may not work in small/large etc models of fossil msdos, but probably such weird models should be forgotten in practice. (regehr-blog:21)
- In C++ systems, calculations like this are often used to find offsets of fields within compound objects (especially when multiple inheritance is involved). Or to make a custom offsetof macro that works for non-POD cases. If constant addresses are required, they have to be non-NULL values. We would usually use something like (WhateverType*)1024 so that the type’s alignment would divide evenly into it, just in case the compiler made any assumptions about pointers being properly aligned. (regehr-blog:26)
- We allocate generated code in separately allocated blocks. When one block runs out of memory, we need to generate a jump to the next block. With relative addressing, this means taking the difference; and when the jump is executed, the subtracted address is added back. (regehr-blog:28)
- Pointer offsets in shared memory segments. (x11:0)
- Results would depend on the size of the objects being pointed to. Malloc results are aligned in a certain way, and if the size of the objects being pointed to are not divisors of that granularity then you’ll get bad results. For example, if p1 and p2 point to separately allocated areas of memory and are of type struct x *, with sizeof(struct x) = 800, then computing p1 - p2 would involve di
vision by 800, and the compiler would throw away the remainder. (x11:1)
- think of a program that has been linked. In theory there are a lot of separate allocation but the linker has smushed them all together into a fixed relationship. Comparison between objects can be common (think of the division of _text to see which are are RO or RW). (x11:3)
- In low-level kernel code there are a number of contexts where one needs to do fairly substantial address arithmetic. To my knowledge, this still works. Most virtual memory systems (not to mention malloc implementations and garbage collectors) rely on being able to do such computations.

However, in most such contexts it will not be clear to a compiler or program checker what a "separately allocated C memory object" is, and therefore the code will generally work even with an aggressive compiler because the compiler won’t be able to do the dataflow analysis needed to break things.

Making it possible for a program checker to usefully validate such code is, as far as I know, an open topic.

Also note that I know specifically of code that uses the address of an arbitrary local variable to approximate the current stack pointer (and then assert that it’s within bounds) -- this is not entirely kosher but there’s no formally permitted better alternative. (x11:5)
- I’ve never tried it, nor am I specifically aware of compilers that will break because of it, but I can’t think of any sensible reason to do something like that. If you have you’ve probably got some fundamental bugs in your understanding of how to use pointers. (xen:2)
- Several common constructs rely on being able to do arbitrary pointer arithmetic on an assumed-flat space.

Some examples:

http://xenbits.xen.org/gitweb/?p=xen.git;a=blob;f=xen/arch/x86/traps.c;h=91701a21415d026917f0ace78b512c712b10d76d;hb=HEAD#l2219

http://xenbits.xen.org/gitweb/?p=xen.git;a=blob;f=xen/arch/x86/traps.c;h=91701a21415d026917f0ace78b512c712b10d76d;hb=HEAD#l3532 (xen:3)

---------------------------------------------------------------

[4/15] Is pointer equality sensitive to their original allocation sites?
For two pointers derived from the addresses of two separate allocations, will equality testing (with ==) of them just compare their runtime values, or might it take their original allocations into account and assume that they do not alias, even if they happen to have the same runtime value? (for current mainstream compilers)

it will just compare the runtime values : 141 (44%)
pointers will compare nonequal if formed from pointers to different allocations : 20 (6%)
either of the above is possible : 101 (31%)
don’t know : 40 (12%)
I don’t know what the question is asking : 16 (5%)
no response : 5

If you clicked either of the first two answers, do you know of real code that relies on it?
yes : 60 (26%)
yes, but it shouldn’t : 16 (7%)
no, but there might well be : 68 (29%)
no, that would be crazy : 46 (20%)
don’t know : 37 (16%)

Wednesday March 09, 2016
Comment

- normally the runtime equality would mean they are the same, but if one point er is obtained from a different process, it may have the same value but point in to completely different memory (different virtual allocation) (main2.2:5)
- This is almost exactly equivalent to the previous question. Again, I would expect pointer arithmetic to work as if every object was allocated from a single flat address space, including testing for equality. And again, I’m aware this is probably technically illegal (but only now you’ve made me think about it). (main2.2:9)
- It seems to me that if the compiler can prove they are different allocation types, that amounts to proving that the pointers can never be equal. I’m not awa re of any compiler that actually does this. (main2.2:18)
- My toy OS kernel does some pointer comparisons with user pointers accross ad dress spaces, or various work while remapping pages. If the compiler decided to elide some checks, that’d make for a "fun" debugging session. (main2:7)
- Never thought about this issue before. (main2:9)
- the pointers could both be null pointers, which will compare equal, even if they come from different malloc calls; I don’t think the compiler can assume mal loc will be successful in order to elide the test (main2:28)
- malloc(0) is the poster-child for this example. There’s much code that relie s on it always being 0, even though that’s unwise (a hold-over from SYS V days). In this case it will compare equal, or the code will expect it to.

However, that behavior is not-standards compliant. (main2:29)

- p1 = malloc(1);
free(p1);
p2 = malloc(1);
same = (p1 == p2)  
--- this test may return TRUE or FALSE and cannot be optimise d out since it depends on how the heap is managed. Just because the two calls to malloc() are at different call sites the pointer values may be the same (eg if the heap happens to return the p1 block back as p2, which it might since that’s good for cache locality). (main2:30)
- Imagine that pointers are implemented as base and offset (see No 3), but the compiler doesn’t enforce range checks. It is important that equality testing ch ecks (base1 + offset1) == (base2 + offset2). (main2:32)
- depends on whether the optimizer runs, I suspect (main2:34)
- The compiler can’t always infer where the object of a pointer was allocated. Even if it can, I don’t believe the compiler should assume the pointers won’t alias (e.g., what about allocating empty vectors?). (main2:35)
- A C compiler can reasonably assume that a pointer generated by taking the ad dress of something on its own stack frame does not alias with pointers gotten fr om somewhere else. Otherwise comparisons should be done on the value.

However note that comparisons known to be always true or false at compile time s hould be a compile error.

[Again the restriction in the standard is related to whether the 8086 pointers i n compact large or huge memory models have or have not been segment normalised. Which is now irrelevant.] (main2:37)
- The code in question is only additional debugging code, not actual applicati on logic. (main2:44)
- Due to the semantics of allocation, it would be reasonable for a compiler to assume inequality (though it may of course compare them e.g. at lower optimizat ion). (main2:50)
- I’m again unclear on exactly what you’re asking. Code would be better than E nglish.

If I understand the question properly, it includes this scenario, in which takin g the allocation into account is not valid: a and b might be equal.

char* a = new foo;
delete a;
char* b = new foo;
if (a==b) ... (main2:55)
- Pointers are just numbers with special significance. How’s the compiler supposed to know they’re from different allocations? And no one does 8086 segmented addressing any more. (main2:66)
- Easy example:

```c
int a, b;
bool x = &a + 1 == &b;
```

Even if b is allocated immediately after a, some compilers constant-fold this to false. (main2:69)
- The rule is that two equal pointers must compare equal. Note that two unequal pointers may also compare equal. I think this is to support segmented addressing systems, which are obsolete.

I note with amusement that your question conflates pointer aliasing and pointer equality which are starkly separate properties. See my oft-misunderstood work "Winner #2" at http://blog.regehr.org/archives/767 where I demonstrate this. (main2:74)
- In LLVM malloc is defined to return new pointers that do not alias any known pointer. The comparison could be optimized away at compilation time, and it should. (main2:78)
- Comparison of pointers to different objects is undefined behavior. (main2:80)
- I can’t tell if by "runtime value" you mean the pointer’s address, or the value pointed to in that address. Pointers evaluate to equal if they are the same address, regardless of what data is pointed. Pointers of two different addresses that point to the same data do not evaluate as equal -- you’d have to dereference and compare the data. I have a feeling this is not what this questions is asking, but I cannot tell what else it is trying to ask. (main2:81)
- A test for a memory manager could have such check. (main2:87)
- Again, what does ‘separate allocation’ mean?

Yes -- the code is any data structure that uses pointer identity. typedef int * HandleType; std::unordered_set<HandleType>.

(main2:90)
- It would be extremely low level and unusual code to be depending on the actual runtime values of the pointers (main2:91)
- malloc/etc can return a previously freed value, so the result is undefined. If both pointed-to objects have been live simultaneously, the operation is safe, but unadvisable. (main2:96)
- depends on aliasing settings
(main2:107)
- why would you ever rely on pointers comparing equal if they were formed from pointers to different objects? (main2:108)
- Why would you compare two pointer values if you know that they’re unaliased. If you need two pointers that are unaliased in a function call (like memcpy), the restrict keyword is your friend. (main2:121)
- If the compiler is certain they’re from two different allocations, I’d be surprised if it didn’t optimize it out. It probably can’t be certain of this frequently though. (main2:126)
- I’ve seen code that uses malloc() as a unique ID generator. (main2:129)
- note that it would be an error if different allocations returned the same value unless there was a free in between or the allocations were on different processors with different memories. (main2:134)
- You may validly use this to see if two objects are the same instance (main2:144)
- For programmes which use dynamic memory allocation, memory reuse can cause false aliasing. Forgoing memory reuse is thought to cause unacceptable system overhead. (main2:147)
- assuming that "derived from the addresses of two separate allocations" means they do not overlap, either of the first two should be possible. if it means any derivation method, it must compare runtime values for equality. (main2:148)
- I know I didn’t click on the first two; but seriously, as with the previous question, there is a _ton_ of code that relies on pointers-are-just-numbers. It
might be wrong by spec, I don’t know; but it has billions of dollars riding on it and compiler authors can’t ignore its existence. (main2:156)
- Again I don’t know what the standard says about this, but I suspect it could be problematic in paged architectures or architectures with multiple base and offset registers. (main2:158)
- Don’t understand the question. How could two separate allocations have the same run time value?! I’m not sure what optimisation the authors are trying to make here, but this sounds like a terrible idea. Each allocation should produce a pointer that is not a subset of current live pointers.

And I assume this is a problem because we compare pointers all the time, including less than style comparison to sort link lists in memory during traversal (for example) (main2:163)
- You could imagine a unit test to check the behavior of your heap allocator. You want to check the invariant that, when all memory is free, the first new allocation always points to the same address:

```c
char* p0;
char* p1;
p0 = (char *)malloc(47);
free(p0);
p1 = (char *)malloc(47);
ASSERT(p0 == p1); // How my malloc is supposed to behave.
```

I do not know of any compiler that guarantees that p0 != p1 just because they have been allocated separately. (main2:165)
- I know no other way to know if both pointers point to the very same object (not to different objects which happen to be equal for example) (main2:173)
- The standard says an object’s address may not be used after deallocation. (main2:174)
- The case is not common enough that compiler writer would invest into such an optimization. (main2:176)
- Questions like these are difficult, as there are lots of embedded systems that do very odd behavior, and their behavior is different than what application developers depend upon. (freebsd:5)
- It’s not valid to compare pointers from different allocations so I would expect that a compiler would be free to use knowledge that the pointers were derived from different allocations to avoid a runtime comparison. (freebsd:6)
- strict aliasing can have far-reaching consequences; using -fno-strict-aliasing is a way to require less thinking. (freebsd:8)
- I assume when directly using a standard new operator, a compiler may make an assumption on the returned value and in a given scope, make a compare of such a pointer against any other pointer a given inequality. That said, if I were a compiler builder, I would at most use this for compile time warnings (unreachable code / expression always false, etc) and not optimize code. The code itself is likely to be naive / inefficient. (google:4)
- pointers from separate allocations must by definition have differing values, so in practice it shouldn’t matter whether the compiler "cheats" by using its knowledge that they "must" be different. (google:5)
- Yes, I see code that relies on pointer == pointer "just working."

I cannot tell from the context of the question whether virtual memory is being considered, or if the question is simply checking for an understanding that malloc(10) + 15 may be the same as a new malloc(10) + 5 (ignoring any allocation metadata, which is implementation-specific). This is a much more severe problem than two pointer values derived from different malloc results comparing equal. Another run might leave them unequal, and depending on conditions, you can get a fault. Fortunately, some memory checkers have the capacity to detect these overflows, usually by increasing distance between allocs.

In the case of virtual memory collisions... In an era where we have enough address space to index the planet’s atoms, it shouldn’t be up to the programmer to worry about collisions with pointer equality between addressing spaces. So in general, while it’s true that the runtime values are simply compared as-is, I would expect that pointers will compare nonequal if formed from pointers to different
nt allocations (except in the case of overflow mentioned above). (google:21)
- Definitely don’t know. Strongly would believe that it just compares the runtime values. (google:26)
- This seems like a question where I’d have to check the behavior of the individual compiler/optimization level. I suspect the possibility of bugs where different optimization levels treat this differently. (google:29)
- Being asked the question is the only reason I would think about such an issue. Having been asked, off the top of my head, the only situation where equality being true would be use of a stale pointer to previously freed storage. Any use of such a stale pointer strikes me as bad. (google:32)
- I interpret the question to mean that pointer comparisons, for two malloc()ed blocks, might be assumed to not compare equally by the compiler, without checking the actual values. I have not observed this optimization but I am aware that there are annotations, as an extension, in gcc and the Intel compiler, and probably others, to say that the returned objects are unique and can’t alias anything which exists, so this seems like a natural extension. I don’t know if the C standard allows such an optimization without annotations -- I suspect it wouldn’t be safe. (google:34)
- If the objects are in different address spaces, the compiler may or may not include an address space identifier (e.g. segment id) in the pointer, and may or may not compare this along with the pointer. (google:36)
- I’d expect a warning if an implementation elides a comparison because it knows the pointers come from different objects. (google:37)
- Aliasing analysis can rely on malloc() never returning the same value twice; assuming neither value was free()d, it can elide such a comparison. (google:39)
- I’ve heard of XOR linked lists at some point, though I haven’t worked with them directly. I think the context was saving memory in GPU drivers.

The concept was that in a doubly-linked list, the previous and next pointers can be XORed together, and you can still traverse the list in either direction. (google:40)
- Two (void or char, not marked with restrict) pointers can always alias and the compiler can never make more assumptions than what it knows from static analysis. I know the standard says differently, but the standard was written decades ago and people need to actually be able work with this stuff in a sane manner to day. C is *the* systems programming language and systems programming requires that you can sanely work with addresses. (For example, I need to be able to define a custom linker script section, wrap it with _mysection and _emysection symbols, and then iterate over objects placed there with for (u32 *ptr = &_mysection; ptr != &_emysection; ptr++) without the compiler assuming that the loop condition can never be false.) (google:43)
- I can’t think of any advantage one might have from making this assumption (that it will just compare the runtime values). (google:45)
- (given I understood the question correctly) I see this frequently, e.g., a global object and a number of dynamic run-time objects. A fast way to check whether the object is the one global is to compare the pointers. (libc:2)
- As for [3/15] there has to be a way to make the compiler just compare runtime values.

But I do know that some useful optimizations fall out of being able to reason from the user-side of malloc, knowing that two different allocations cannot alias, which implies their pointers cannot be equal. (libc:6)
- I know in practice that compilers will optimize comparisons of pointers to separate allocations to false. However, if optimization fails, the runtime values will be compared, and you can get either behavior. (llvm:2)
- Unknown what the language says, I assume the value of the ptr to be identical to the value of (void) ptr. If you want to compare what the pointer points at you dereference the ptr. (llvm:3)
- There really can’t be anything "special" about C pointers just because they are returned by a heap allocator. For one thing, the heap allocators are probably written in C and must properly free an allocated block no matter how much arithmetic is done on the pointer between the allocation and the free (as long as the value ends up being the same thing returned).

A common technique in operating systems when they need a temporary buffer it to keep a small buffer in the stack frame and only use the heap allocator when the
buffer in the frame is too small. In this case, the free is doing to look like
"if (p != framebuf) then free(p)". This is done because heap allocators in a ke
rnel environment need to grab locks and that make them very slow. (llvm:5)
- There is real code that uses kmalloc(0) to generate unique cookies to use as
identifiers (extremely bad design, but it does exist.) Also there is code that
allocates structures that may be optimized to a size of 0 depending on compile t
ime options. Pointer equality is then later used to test for identity. (llvm:6)
- Functions like memmove() do void* pointer comparisons to determine whether t
he given memory regions overlap. In the case of pointer equality, this function
can return immediate (no-op) as a fast path. (llvm:7)
- A pointer lvalue containing a dangling pointer is indeterminate. Using indet
erminate values might as well be undefined behavior, and going out of bounds is
undefined behavior. If this does not happen, then the comparison of pointer val
ues is indistinguishable from any technique enhanced with allocation site conside
rations. See also: http://trust-in-soft.com/dangling-pointer-indeterminate/

But then again, the C standard does say that &a+1 and &b can be equal if object b
happens to follow object a in memory. That clause may or may not allow the com
piler to be inconsistent in the value it gives to &a + 1 == &b. I don’t know of
any compiler that would allow itself to be inconsistent, but you cannot put it p
ast any of the optimizing ones. (regehr-blog:0)
- I can’t think of any situation where this would matter *and* the compiler co
uld realistically be sufficiently smart to perform the optimization. (regehr-blo
g:12)
- I don’t work with dynamically allocated memory, so no relevant experience. (regehr-blog:14)
- The question as worded is ambiguous -- if there is no "free" of the original
ly allocated object in between, then the pointers must be distinct statically (a
nd by runtime comparison), and compiler is allowed to exploit that. A "free" of
the first object in between could make the pointers equal dynamically, probably
compiler is not allowed to statically make assumption of them being not equal. (regehr-blog:16)
- Not a scenario I can recall any firsthand experience with. (regehr-blog:17)
- Use of a pointer to an allocated block is undefined if the block has been fr
eéd, that includes comparing two copies of the pointer to each other.

So in a scenario where you allocate memory, then keep the pointer around but fre
e the memory, then allocate new memory which happens to have the same address --
there is no legal way to compare the old pointer to the new one anyway. If nei
ther object has been freed, their addresses won’t be the same anyway. (Even for
zero-byte allocations, most compilers return at least 1 byte of storage.) (rege
hr-blog:26)
- gcc’s "malloc" function attribute allows the programmer to specify that a fu
ction behaves like malloc -- that is, its pointer return value does not alias a
nd thus offers optimization opportunities. (x11:2)
- This is gets nasty with PIC references and lazy evaluations especially for f
unction values. (x11:3)
- As far as I know current compilers aren’t capable of this level of whole-pro
gram analysis yet. (But maybe I’m behind?)

Outside of the memory management context, I would be surprised to find code that
relied on being able to do this.

In the memory management context... I don’t know, I’m not sure "allocation sites
" is even well defined. If I go through some data structure, fetch some values, com
pute on them, and then return a value from malloc, that value is itself an al
location, but within the malloc code where’s the allocation site? And if I later
(e.g. in free) compare the pointer value to other values I’ve fetched from my d
ata structures, is that a pointer value from the same or a different allocation
site? The question isn’t even particularly well formed.

Note that there’s another related topic, which is: can a pointer that comes from
allocation site A compare equal to a constant pointer value that someone synthe
sized out of nothing? (such as the popular (void *)0xdeadbeef)

I can imagine a compiler writer trying to optimize out tests of that form, and t
ry that *will* break things, e.g. the POSIX constant MAP_FAILED. (x11:5)
- This whole "compilers will assume code is like X and do something unpredictable and irrational if it’s not" thing is something I’ve got a tiny amount of exposure to; so since you ask the question, I have an inkling many compilers will do something unpredictable and irrational as a result, but I don’t think about it in my day-to-day programming.

In any case, as I said above, I can’t think of a sensible reason one would want to compare pointers that came from different allocations; if you’re doing that you’re probably doing something very wrong already. (xen:2)

[5/15] Can pointer values be copied indirectly?
Can you make a usable copy of a pointer by copying its representation bytes with code that indirectly computes the identity function on them, e.g. writing the pointer value to a file and then reading it back, and using compression or encryption on the way?

Will that work in normal C compilers?
- yes : 216 (68%)
- only sometimes : 50 (15%)
- no : 18 (5%)
- don’t know : 24 (7%)
- I don’t know what the question is asking : 9 (2%)
- no response : 6

Do you know of real code that relies on it?
- yes : 101 (33%)
- yes, but it shouldn’t : 24 (7%)
- no, but there might well be : 100 (33%)
- no, that would be crazy : 54 (17%)
- don’t know : 23 (7%)
- no response : 21

Comment
- Marshalling data between guest and hypervisor (main2.2:0)
- can’t think of an application that requires it. The memory would only be valid within the same process, so why not make direct copy (main2.2:3)
- it’s usually a bad idea to write pointers into files and then read them back, since that memory may no longer be allocated. (main2.2:5)
- (Yes, as far as it dereferenced only within the same process.)

1) Storing a bit or two within an (aligned) pointer (and masking them out when going to dereference).
2) Compression of really big data collection, having only a cache of some uncompressed members of it.
3) Writing a debug allocator (wrapper on top of malloc()/free()/realloc()) which just stored some offsets in its internal structures to save some space. (main2.2:6)
- Various software that passes pointers around as opaque objects, including to other cores that cannot address that space. (main2.2:8)
- This is the xor-linked-list question. It should work. (main2.2:9)
- I’d expect massive breakage of common software from any system that didn’t have this property. (main2.2:18)
- Ought to work with a uintptr_t or char* aliasing. (main2.7)
- That’s used e.g. to pass handles to objects to other processes (using sockets, not files), but this is an important use case. (main2.8)
- I’ve written code for a JIT that stores 64-bit virtual ptrs as their hardware based 48-bits. This is a valuable optimisation, even if it’s not strictly OK. (main2.9)
- Same as the interpret cast. Do whatever you want, but get back the same original thing and it’s OK. (main2.10)
- Won’t work with CHERI (main2.16)
- Pretty sure this doesn’t work on CHERI. (main2.18)
- It does interesting things to the escape analysis. (main2.19)
- there is the double-linked-list xor trick, which is horrible but real (xor t
he forward and backward pointers together, then xor while traversing in either d
irection to recover the desired next pointer)
(main2:28)
- There are many instances where code bcopy’s a structure and expects the point
ers to survive. The way the question was worded, however, makes me think that it
is a trick question. (main2:29)
- Windows /GS stack cookies do this all the time to protect the return address.
The return address is encrypted on the stack, and decrypted as part of the fun
cion epilogue. (main2:30)
- I can imagine a compiler implementing a pointer as a handle (pointer to point
er) to assist with garbage collection. The additional redirections may be hidde
n from the programmer. (main2:32)
- the compiler is at liberty to do what it pleases with the pointers; we shoul
dn’t be copying them around with bitwise manipulations but treat as black boxes
(main2:34)
- The identity function is the identity function! (main2:35)
- You can go much stronger than that. Many security mitigation techniques rel
y on being able to XOR a pointer with one or more values and recover the pointer
later by again XORing with one or more possible different values, (whose total
XOR is the same as the original set).
(main2:37)
- Only for NULL (main2:45)
- Isn’t this exactly the use of intptr_t, which is C99 (possible optional)? (m
ain2:48)
- The current Julia task−scheduler does this, by way of copying a task’s stack
into a buffer, and copying the buffer back to the stack later. (main2:49)
- This question seems absurd to me. It is very common to take a pointer with s
ome known alignment requirement and combine it with flags or other info in the l
ow-order bits, which would need to be masked off again to recover a usable point
er. E.g. the MMU translation table register of ARM processors is structured like
that. (main2:50)
- Well, it would work as long as you don’t relaunch the program. It’s not cle
ar what scenario(s) this question is envisaging. (main2:51)
- Surely there’s VM code somewhere that does this kind of thing? (main2:54)
- If you use lossy compression and encryption (e.g. methods that assume text o
r Unicode) you might not get the same bytes.
The file of bytes certainly won’t help if the program with the pointer ends.
Sneaking around the compiler’s back like this might invalidate some optimization
s. (main2:55)
- Code that does this is usually a security exploit waiting to happen. It’s a
lose a pain to maintain. (main2:59)
- I was under the impression that casting to intptr_t and back is defined? But
what do I know. (main2:61)
- I used it in games for in−place loading. (main2:64)
- Some compilers require the computation of the pointer to somehow depend on t
he original pointer -- you can round−trip through a file, but you can’t just gue
ss the address, even if you guess right (for instance, if you ask the user to ty
pe in a number and assume it’s the pointer, and the user gets the number from a
debugger, that will not work in practice). (main2:69)
- The only reliable way to do this, that I know of, is to convert it to an imp
lementation defined representation using (f|s)printf %p on the original pointer
cast to void * then reading it back with (f|s)scanf %p then converting it back t
o its original pointer type.

As you said, you’re not asking what the standard permits, but I don’t like to as
sume what compilers do. (main2:70)
- I would be interested in learning what practical complications could arise f
rom this, during the lifetime of a single instance. To say this shouldn’t be use
d would seem to imply that all pointers should be treated as volatile. Maybe I m
isunderstood this question? (main2:72)
- As long as the object lives between writing the pointer and reading it, and
has not been reallocated. (main2:73)
- Pretty sure this is valid behaviour. We go out of our way to support this.
Well, okay, it depends how indirectly. If you want to be completely loopy, this won’t work in our compiler:

```c
bool isThisIt(uintptr_t i) { return i == 0x12341234; }
void *launder_pointer() {
    int stackobj;
    for (uintptr_t i = 0; ; ++i) {
        if (isThisIt(&stackobj + i)) { return (void*)(i - 0x12341234); }
    }
}
```

because we may return false for every call to isThisIt() even though I think it’s technically valid. We generally forbid guessing the addresses of values where we’re allowed to pick the address (ie., we fold "&stackobj == (void*)rand()" to false), but we didn’t account for the case someone tries the entire address space in a loop. Don’t care.

Taking the pointer and capturing/escaping it is supported, we assume it may come back in from anywhere in the future, including by being typed in at the console . (main2:74)
- Probably not allowed by the standard (main2:79)
- I don’t know the answer. I did some experiments and it seems that the codes work just fine. (main2:80)
- This is assuming that the copy exists in the same address space where the original pointer was created. (main2:83)
- In LabVIEW on Windows, you can call C DLLs. I presume you can do likewise on other systems. It is very natural to call a C function that produces a pointer, store it in an integer, then pass that back to another C function. The correct way is more work: return an identifier that the second C function can then use to look up the pointer. I’ve seen and written code that does it the right way; I may have seen the wrong way, but can’t remember. (main2:85)
- Example I know: we copy input data to a function into internal format, and don’t destroy the internal copy on exit from the function. So that second call to the functions with the same pointer will cause reuse of the internal copy from previous call and avoid copying overhead. (main2:87)
- Hard to tell whether it shouldn’t... there are terrible cases like creating a string from a pointer and parsing it back just because that’s the only way the engineer could get their work done with a given set of APIs, but then there are more reasonable (nevertheless crazy) cases like a xor linked list (NB that requires Q3 to be true). (main2:90)
- You can copy a pointer by copying the bytes of its representation.

Of course that only works for as long as the object pointed to remains allocated . (main2:92)
- Again, not standards compliant but a compiler which enforced this part of the standard would not be fit for purpose. (main2:93)
- The standard allows you to access the representation of a pointer (or any other object) via an unsigned char * (representation of types, clause 6.2.6)

realloc() needs this. (main2:94)
- Half of those questions seem to be special cases of "does your C compiler include a compacting garbage collector?". (It does not.) (main2:96)
- I suspect bit fiddling in "optimized" swap code (main2:98)
- I may be out of date WRT the latest spec but I believe C99 6.2.6.1 enables this practice and that nothing forbids it.
Obviously it will break when using a conservative GC. (main2:100)
- I’m assuming the reference to "copying its representation bytes" excludes casting to uintptr_t. This is still very common in embedded systems, especially for pointers to memory mapped flash/eeprom. (main2:106)
- Only yes, if it’s not a shallow copy of something with pointers to stack stuff (or VLA’d or alloca’d arrays).
- The xor two way linked list trick falls into this category, right? As/400 and other segment or descriptor machines nuke this, again. (main2:111)
- I don’t see anything wrong with this, as long as you make sure not to free any pointers in between the write and the read. (main2:121)
- Within the same process writing a pointer address and then later reading it should be reasonable. (main2:127)
- I wrote an RPC system that serializes pointers using intptr_t. (main2:129)
- memcpy on a struct containing pointers counts here, right? (main2:130)
- In embedded systems, it is common to have statically allocated regions. So saving and loading a pointer back will work. (main2:140)
- Many programmers take advantage of the fact that a pointer has the same size as an int.
- For some simple programmes including the same process in different runs) this will not work.

I think the requirements are stronger than uniform memory architecture because e.g. in the presence of virtual memory, arbitrary page tables (the programme could remap it’s memory so the same address refers to a different object) are needed to resolve an address value to the right object. Does ”representation bytes” st and for arbitrary page tables? I think it doesn’t in the sense that page tables aren’t portable. (main2:147)
- this is dumb to do, but if the address is written and read by the same instantiation of the process, this should function. (main2:148)
- Absolutely. Tons and tons of code. If ”it shouldn’t” by spec, again, spec here is just out of touch with reality. Pointer mangling and demangling happens all the time. Look at the internals of any interpreter for goodness sake: tagged pointers are doing this on _every value_. (main2:156)
- I am not sure what the applications for this might be, since the pointer values would only be relevant during the specific run-time of a given program (especially when factoring in modern operating system protections like address space randomization). (main2:158)
- Of course, you have to be careful of the aliasing rules. (main2:160)
- Funnily enough, I’ve seen this used for copy protection. (main2:163)
- If your pointer’s value is the address of a global variable or a function, I do not see why this would be a problem, providing that you know for sure that the compression-decompression or encryption-decryption scheme is lossless. (main2:165)
- Gil+Viktor first reaction: they don’t have any reason to believe that compilers won’t do optimisation based on alias analysis. But then (when prompted) they think maybe the compiler does treat the result as a pointer about which alias analysis knows nothing. Any marshalling/unmarshalling relies on this. (main2:172)
- Any kind of memory dump (for any purpose) does that. If one want to safely reuse the pointer, one has to ensure that all the data are exactly at the same memory though, which can sometimes be difficult. (main2:173)
- xor doubly-linked list
- cast from uintptr_t ...
- BLOSC (http://blosc.org/) does something like this. It compresses data stored in RAM with the goal of reading compressed data from RAM into L1 cache faster than an uncompressed memcpy. If pointer values can’t be copied indirectly, then BLOSC users are in trouble. (freebsd:0)
- You gave me a wiggle room in this question. I am considering a mmap SHARED segment as a file, since this may be backed by such a file. If the pointers are shared between processes, it is possible to do this. (freebsd:5)
- If an executable serialised a pointer value to a file and then the same executable read the pointer back as the same type then I would expect the resultant pointer would be valid. But I’m not certain and I can’t think of any valid reason for doing so. (freebsd:6)
- This is clear as a consequence of Representation of Types. (gcc:2)
- Normal for code that interfaces to other languages, e.g. with Java JNI storing a C pointer in a Java jlong object. (gcc:3)
- I’ve seen this technique used extensively for message passing and queues in shared memory regions.
This only works when the virtual memory map of the writing and reading code have the same addresses for the memory which the pointer points to. (google:1)
- I am unsure what this question is asking? If a pointer ‘value’ can be stored? Of course, you can take a pointer’s value (say, use intptr_t) and store it in any format suitable for storing integer values (ascii, base64, be/le binary etc) and read it back. No different from stuffing or copying a pointer value anywhere else, in practice you will most likely never do this as persisting pointer values is rather useless unless you’re a debugger / core dump. (google:4)
- I saw some code that saved space by storing only the lower 32 bits of 64-bit pointers, because the data storage requirements were vastly affected by whether all 64 bits of a pointer were stored.

The code has since been fixed; I believe the current scheme is to allocate all data as part of an ever-growing deque>, and then store 32-bit indices into the deque. (google:7)
- Swap file (google:9)
- A classic example of this behavior is the pointer XOR technique for reducing the footprint of linked list nodes.

I’m not certain if this is actually a good idea or not in modern software. It would screw up any prefetching optimizations that might be available, but on the other hand it might make the node able to fit in a single memory I/O or make a difference in cache behavior.

There’s also the "XOR swap" technique, again still extant (mostly as the answer to trivia questions), but this one should be more strongly discouraged against because there’s no benefit on modern architectures. (google:11)
- I don’t see any reason why you would think this is bad. I can think of applications, (especially when memory is a limit factor) where this approach would be mandatory (or at least easier to code) (google:13)
- E.g. memcpy of a struct containing a pointer works (assuming alignment etc.) (google:14)
- garbage collectors (google:16)
- I’m assuming single run of the program (google:19)
- does not work if the pointer points to e.g. a struct that has other pointers (you’d need to do some recursive serialization) (google:20)
- Consider implementations of string, where the string pointer is a char* which (as usual) indicates the beginning of the string’s contents in memory, but the word before this string is the length of the string.

Only the pointer to the string itself is kept handy, so that puts(string) will write the contained data, while printf("%d\n", ((int*)string)[-1]) will print its length, and along that same line, the free must be performed on (((int*)string) - 1).

The GNU std::string implementation relies on this. (google:21)
- I believe you are allowed to store pointers in intptr_t’s and then do whatever with them. (google:22)
- I’m pretty sure this will always work for x86 (google:24)
- It’s a terrible idea, but sure, why wouldn’t it? A pointer is just a number ...
- We had to swap parts out due to memory peaks. The pointers were to stuff that was not swapped. (google:27)
- Assuming you mean the same execution of a program is writing the pointer.

I can also imagine a shared memory map involving multiple threads or multiple processes accessing the same memory, where pointers may be passed from one to the other. (google:31)
- It seems likely to work, but it sounds too stupid for words. (google:32)
- for what: Sending messages to itself over a pipe (to allow use of select/poll as a single event management interface). Also memcpy() on structures.

If this is not actually valid I think there are a lot of programs which will need to be rewritten :/ (google:34)
- Pointers may be stored in some arbitrary data chunk, which doesn’t know the
kind of data it is storing (such as a uint8_t * buffer), and then extracted late
r. (google:35)

− Most common way to do this is to memcpy a structure containing a pointer. (g
oogle:36)

− Gtk and glib uses it e.g. to store pointers in integers or vice versa for so
me kinds of callback parameters. I don’t like this practice, but it seems common
, and uintptr_t seems to exist for this very purpose. (google:37)

− This happens when pointers are handed around in cases of shared memory. For
example, in a system without a protected memory model, processes can pass pointe
rs between them, and sometimes those pointers might be compressed into a blob, s
erialized to a string, etc. (google:38)

− Seems like this might be common in IPC code, e.g., sending a pointer or an o
ffset to shared memory over a local socket. I haven’t personally seen that, but
I would expect it to work. (google:40)

− This seems to ask whether you can serialize and deserialize data. Assuming t
he data pointed to supports that (i.e. you know exactly how many bytes to serial
ize, and deserializing the serialized data returns exactly the same data) there’
 s nothing that prevents you from writing it to disk, allocating enough memory fo
r another pointer, and copying the data into there. (google:41)

− depends on if this is a pointer to an object on the stack or not. Compiler m
ay do liveness analysis, and re-use the stack bytes for some other purpose. (goo
gle:42)

− On any sane architecture (e.g. single uniform linear address space, not 8086
), yes. This is important for systems/embedded use cases. I need to be able to l
oad a new program somewhere and pass addresses to system-global data structures
to it. (google:43)

− yes (for what?): Storing pointers in [u]intptr_t type integers. (google:45)

− Sure, lots of code does that. Doesn’t work if you are using a conservative
GC, though. (libc:1)

− I used this in an internal program pipe to exchange object references betwee
n application parts. Of course, this only works within one program. (libc:2)

− i consider most use cases non-portable (implementation specific hacks), but
they are common.

pointer tagging (this makes lots of assumptions about pointer representation and
its mapping to integers, but often used to store information in the bits that a
re known to be fixed)

xor linked list (never seen it in actual use)

xor function pointers with secret (function pointers at fixed address in writabl
e memory can be dangerous if an attacker may overwrite their value, so there are
hardening techniques which store an "encrypted" version of the pointer and "dec
rypt" it for dereference)

some applications mmap a file with stored pointers at fixed address so the inter
nal pointers work.. this should be considered a non-portable hack.

(libc:3)

− Another sign of bad code smell. It will probably work on most systems becau
se the pointer representation is just a memory address, but it may seriously int
fer with optimizations, and the compiler could break it and be justified in d
oing so. (libc:4)

− Probably all memory management code relies on this. (libc:6)

− For some forms of pointer checking extensions doing this will cause the poin
ter to lose its associated bounds information, which is kept separately from the
actual pointer bits. (llvm:1)

− It works as long as the memory pointed at is not freed (llvm:3)

− It’s obviously not going to work if the storage gets freed.

You are describing an ELF executable. pretty well. (llvm:5)

− Again, this requires casting between the pointer and an appropriate data typ
e like size_t. It is not inherently safe, but a pointer (memory address) is just
a series of bits at the end of the day. (llvm:7)

− This indirect copying might start by converting pointers to uintptr_t, after
which what happens is implementation-defined. One could hope that this kind of
use is implicitly allowed on conventional compilers and that an implementation t
hat precludes this use would document it. (regehr-blog:0)
- I think this depends on what one means by "usable copy": You should be able to get a copy that you can dereference to access the same memory location as with the original. But if you tried to use == to verify that you got a copy, that might not always produce true. (regehr-blog:2)
- I assume we’re not asking about storing a pointer’s value in a file for use across invocations or by other programs, which any given tutorial will warn against. If we’re staying within a single invocation, then this will work, as long as the pointed-to memory will still be there when the reference is rebuilt. I’m not sure what compression or encryption have to do with this, but I assume the rebuilt reference is decompressed and/or decrypted. (regehr-blog:7)
- I have heard others stating that this is a valid way to obtain a pointer in C, and I can envision shared memory scenarios where that’s plausible. (regehr-blog:9)
- I’ve definitely done it while debugging, with identity = printf() -> manually type pointer in elsewhere — does that count? (regehr-blog:10)
- I think this is totally OK in C11, assuming you memcpy() the value out of an into the pointer, or cast to and from intptr_t or uintptr_t. I can imagine it being useful either as part of some temporary memory compaction scheme or to send a pointer through a (not-well-written) library to a callback function. (regehr-blog:12)
- I assume this refers to using the file to pass a pointer value from one invocation of the program to another.

If instead were careful to only read the file from the same invocation that wrote to it then I suppose it would work. (regehr-blog:14)
- Conversion between pointers and (u)intptr_t is required to be lossless. (regehr-blog:16)
- I don’t remember exactly what it was, but I recall reading recently about an application that would serialize a region of memory directly to a file and then restore it via mmap(2) to a fixed address, reusing pointers within it without any sort of conversion or translation. (regehr-blog:17)
- A tool I maintain (ab)uses this behaviour: the storage of the objects is never reclaimed because the tool itself is a "one-shot" (it does not last long like a server).

In the process it may require to "serialize" some objects into a string representation, and currently it does by printing the address of the pointer and later on parsing it again to get access to the object. The reason behind the serialization process is beyond the scope of this comment.

This is just a convenience since an identifier (of a table) could be used instead. (regehr-blog:18)
- for example to move the pointer between threads as raw bytes array. (regehr-blog:21)
- It sounds crazy, but may be used in some distributed system or even multithreaded app (regehr-blog:22)
- If you use a garbage collector, of course this would break it. So would other tricks that obfuscate the values of the pointers (e.g. xor-ing together the next and prev pointers in a doubly linked list).

Most systems aren’t using a garbage collector, and as long as you don’t actually free the storage, obfuscating and later recovering the pointer to it should work fine in practice. (regehr-blog:26)
- Accessing data in shared memory segments with MAP_FIXED. (x11:0)
- gnu-emacs undump. ugh. (x11:3)
- It’s not uncommon to find code that writes random stuff out to a temporary file and reads it back. Sometimes that stuff includes pointers.

Also, there’s a lot of code that uses memcpy or memmove to shift around values in arrays, and some of those values contain pointers. Or did you not mean to include that?

And I should point out that every virtual memory system writes out pages containing pointers and expects to be able to read them back again, and some virtual memory systems do compression and/or encryption on their swap space.
Then consider the following horror from my quotes file:
Creative C code of the week:

```c
sprintf(result7, "%d", fclose((FILE*)atoi(libp->para[0])));
```

That’s from a package called "utilisp", which is full of such rubbish (which I believe to be compiler output, but even so...) Q.v., but not viewable on a full stomach. (x11:5)
- people do this with mmaped files at fixed locations and then using it as memory. maybe gcc does this (x11:6)
- One thing someone might do is to memory-map a file in a specific address in memory, and have pointers within the file/memory that point to other places inside the file/memory. That way you could store a complex data structure on disk without needing to serialize it on save and re-construct it on reading it.

I’m not aware of any programs that do it, but it’s the kind of thing I would expect to be able to do.

Storing the results of malloc() in a file seems crazy. (xen:2)
- Stack traces, instruction emulation. (xen:3)

[6/15] Pointer comparison at different types
Can one do == comparison between pointers to objects of different types (e.g. pointers to int, float, and different struct types)?

Will that work in normal C compilers?
- yes : 175 (55%)
- only sometimes : 67 (21%)
- no : 44 (13%)
- don’t know : 29 (9%)
- I don’t know what the question is asking : 2 (0%)
- no response : 6

Do you know of real code that relies on it?
- yes : 111 (35%)
- yes, but it shouldn’t : 47 (15%)
- no, but there might well be : 107 (34%)
- no, that would be crazy : 27 (8%)
- don’t know : 17 (5%)
- no response : 14

Comment
- You need a cast, I think. (main2.2:0)
- comparison of different pointer types should yield a warning, if not an error on normal compilers (main2.2:3)
- code that masks the type of data (with void*) may use == to compare a void* to an actual type* (main2.2:5)
- Yes, as far as both types are compatible with the respect to the pointer aliasing rules. (main2.2:6)
- I’d assume that this would result in UB (main2.2:7)
- It may need to be cast to avoid compiler warnings, and in some cases errors, but pointers originally of different types do get compared. (main2.2:8)
- If you cast everything to void* then it doesn’t matter what the original types of the pointers were. I would expect:

```c
void* p = malloc( 10 );
int* a = (int*)p;
float* b = (float*)b;
assert( (void*)a == (void*)b );
```

To work. (main2.2:9)
- Expect to be able to == compare (void*) to more specific pointer types like (int*) (main2.2:15)
- I actually don’t know what the standard says about this (main2.0)
- only acceptable when comparing struct pointers where the first element match
es the type of the other pointer, otherwise "but it shouldn’t". (main2:3)
− The code probably converts the pointers to void* or an integer before compar-
ing, but apart from this, code like that should very common in maps or hash tabl-
es. (main2:8)
− Presumably this violates strict aliasing. (main2:9)
− Depends on strict-aliasing flags? I think LLVM TBAA might optimise this sort
of check away? (main2:12)
− Using Casting, sure. (main2:14)
− Another reason for fno-strict-aliasing... (main2:23)
− I imagine under some optimisations that this will break horribly. (main2:26)
− This is common in runtimes. With casting, it is safe in practice in all com-
piers I have seen. It is likely not safe under all optimizations. (main2:27)
− Compilers will at least warn about this - have commonly seen explicit casts
to compensate, can’t recall seeing a "naked" comparison of different types (main2:28)
− You need to cast first. (main2:29)
− Might be more common with pointers to unions in play, and need to explicitly
cast to a common type (eg void*) to calm the compiler. (main2:30)
− Pointers may be forced to align with multiples of the size of the object the
y refer to. e.g. Some architectures ignore low-order bits in an address dependin-
g upon the size of the object being referenced. (main2:32)
− Question is a little ambiguous. I assume casting is involved. (main2:33)
− You don’t mention == implicit casting when comparing with void* (main2:34)
− Surely any heterogeneous collection data structure has to do this. (main2:35)
− When first field in a bigger struct is a smaller struct. Code could compare
pointers to those different structs. (main2:38)
− Function pointers don’t have to be the same size as data pointers. Since yo
u didn’t exclude functions as one of the types, that’s the most obvious place wh
ere comparison is not a good idea. (main2:48)
− I have no idea what liberties compilers take with strict aliasing rules enab-
led, hence my answer "don’t know". When compiling with strict aliasing disabled,
it had better work (apart from needing a suitable cast of one or both pointers
to avoid a type mismatch). (main2:50)
− I was under the impression that compilers are free to assume that pointers o
f different types don’t alias (as long as neither is char* or void*? I forget),
so I wouldn’t be surprised to see compilers optimizing out the comparison -- bu
t I’ve never actually tried this. (main2:54)
− If I saw code that was doing this, I would consider it high priority to fix
it. (main2:59)
− I’m presuming here you mean "without casting to a void *". (main2:66)
− struct hdr;

struct whatever {
  struct hdr;
  /* more... */
};

(struct whatever*) somehdrptr == somewhateverp (main2:67)
− Depends on compiler flags. (main2:69)
− The equality operator constraints are such that both operands are pointers t
o qualified or unqualified versions of compatible types, or one is an object poi
nter and one is a qualified or unqualified void pointer. Your question doesn’t s
pecify unambiguously whether or not such a constraint was met, though the exampl
es suggest not.

(main2:70)
− In C++ this error is caught at compilation time. I suppose in C it works th
e same way. (main2:78)
− The type of the pointer doesn’t matter. What matters is where the pointers p
oint to: if the pointed-to objects are different, the behavior is undefined. (ma
in2:80)
− May need to use casting or union types, but this is essentially what goes on
with pointer tagging/immediate types in virtual machines. (main2:83)
− example: Bitwise comparison of floats by addressing these as integers (main2:

87)
- Type punning is a thing in low-level systems. But I find that I don't always want the full power of that. The regular == should be strict, but then there should be ==w, a weak equality that does not require strict aliasing.

There are different audiences that want different semantics from C. Some people want high-level C, and strictness does not really hurt. Other engineers want the semantics of a portable assembler, and that's understandable, but applying strict semantics to such code can lead to miscompiles. The worst thing is a highly optimized codebase, like a language runtime, that wants both semantics, but in different parts of the project. (main2:90)
- The compiler probably wants a cast to avoid a warning, but otherwise I would treat this as OK (if perhaps questionable style depending what the code was doing). (main2:93)
- Strict aliasing problems. (main2:94)
- You'll need to go via 'void *'. (main2:100)
- A valid case I see is for object code in C, e.g:

```c
struct A {
    int foo;
};

struct B {
    struct A parent;
    int bar;
};
```

A pointer to a struct A could be compared to a pointer to a struct B (and it should work!).
- This will provoke at least a warning, unless you mean that the pointers are cast to e.g. void * before comparison. (main2:103)
- Not always crazy if you're doing fancy casting and stuff with unions. (main2:112)
- Void and char only, but other objects don't alias. (main2:115)
- lol (main2:119)
- I think it should work, but I know changing a pointer to point at a differently sized type is undefined behavior, iirc. (main2:121)
- I say yes but generally such code is going to go through casts to void * or char *. I'm not clear about how this interacts with the strict aliasing rules.
- yes, if you cast the pointers to a void pointer. (main2:124)
- Compilers normally reject direct comparisons but it is common to cast to char* or void* to compare pointers of disparate types. (main2:127)
- "yes" means "will compile it" - but it may not do what you expect.

will produce different results on big-endian versus little-endian architectures.
- Seems illegal without reinterpreting to void* or ptrdiff_t but I am sure many allow it. (main2:134)
- Unions are the recommended method to do polymorphism but not easy for clients to extend. (main2:138)
- Unions, C polymorphism and inheritance (main2:148)
- Via cast though (main2:150)
- In the strict sense there are a bunch of these comparisons that the compiler will flat out reject, statically, unless you cast. If you include casting to union or void* or whatever, then again, this happens constantly.

I feel like these questions perhaps ought to be phrased as "...with casts", and separately without. And anyone interpreting the responses should understand that the MO of a huge percentage of C coders and C libraries and C macros is "keep casting away C semantics, in the direction of assembly-language semantics, until it works". (main2:156)
- It is possible to compare pointer types by casting but modern compilers should provide type warnings here. In general on most common modern architectures pointer types will be physically compatible, that is they will have the same length, which may not have been true historically. (main2:158)
- Any dynamically-typed language interpreter does this. Each structure representing a type will have an identifier as its first member (eg. See lua language). I would expect pointer equality to also follow this.

We do union like operations (by casting) a LOT in this industry. (main2:163)
- Gil: they don’t use alias analysis for pointer equality. But if it knows provenance is difference, it’ll be false. (main2:172)
- Casting the pointer before making the comparison would be safer. (main2:173)
- Type-based aliasing analysis assumes they’re unequal (except signed char == unsigned char). gcc gives warning about it. (main2:176)
- With -fno-strict-aliasing (the most common setting in my experience), the compiler will just compare runtime values. With -fstrict-aliasing (rarely used in my experience), the pointers will compare nonequal. (freebsd:0)
- It is likely that code that marshals data, like ASN.1 parsers may use this for some reason. (freebsd:5)
- This definitely used to be possible (just look at 6th Edition code) but C has been becoming steadily more typesafe and strict. I don’t know if it’s currently valid or whether there’s current code that relies this behaviour. (freebsd:6)
- I’m sure that people are aliasing the first member of a struct type, somehow. (freebsd:8)
- Fails on systems that use a different representation for char* pointers, such as DECSystem 20 and some embedded processors. (gcc:1)
- Assuming they’re both converted to a type that can represent both, e.g. void * or char*.
  (gcc:2)
- I would expect at least some compilers to produce a warning. (gcc:3)
- Provided you cast to void pointers, you can compare any pointer to any type. If it makes sense depends very much on your code. You may have union data and want to check for aliasing on operator= assignment or any such? Most times you use this there is a likely code smell. (google:4)
- Wouldn’t that cause a compile error? (google:7)
- The only reason I say "it shouldn’t" is because it makes compilers throw warnings, and so any reasonable code will have explicit casts to shut it up. The actual behavior (even if hidden behind a cast) can be seen in the wild -- I know Python does essentially this as a way to implement polymorphic structs in C.

This DOESN’T work in C++ because the compiler escalates that warning to an error, which means that any code that wants to compile in MSVC *must* provide explicit casts because MSVC is a C++ compiler, not a C compiler. (google:11)
- There are a lot of examples of this, in particular in libc, or possibly implementations of vtables.
  (google:13)
- A hashtable that uses "special" objects. (google:14)
- I assume this would not always work because it would violate the strict aliasing rule, which compilers are known to take advantage of. (google:18)
- I can’t actually name a specific example, but this test might be useful in the event that a function is likely to be passed members of a union. (google:21)
- Compiler error. (google:23)
- This will work unless you are on an architecture that has different pointer representations for different types. (google:24)
- It works OK if they are the same size. (google:27)
- This ought, I think, with a sufficiently strict compiler, throw a type error. But that can be gotten around with casting.

In real code, I’ve seen shifting between pointers to a struct and pointers to its first element. I believe this behavior is codified in the spec. (google:29)
- Ok for T* == void*, with different types it will trigger a warning. (google:30)
- I can see an application involving unions. (google:31)
- This works frequently in practice but it’s guaranteed to do something useful. Possible reasons are segmented memory and typed pointers. (google:34)
- Comparing pointers to different types is a symptom that there’s something wrong with the program. (google:36)
- function and data pointers are not always comparable with predictable results (example: memory models with 16-bit function pointers and 32-bit data pointers, e.g. common on MS-DOS).
Real world code tends to compare `foo*` to `void*` a lot. Even seeing this a lot where `foo` is a function pointer type and this really shouldn’t be done then. (google:37)
- I’ve seen comparisons between pointers where one refers to a structure and another to a member of that structure. This is reliable only with packed structures. (google:38)
- I’ve seen this used for object systems implemented in C. If one struct is a "sub-class" of another (they had the same initial members), this would have been considered okay. I think `-fno-strict-aliasing` or something equivalent was needed for that. (google:40)
- You at least need to cast the types to the same type. I see this mostly when unsigned and signed integers are compared, or when model values that are integers are somehow used in user interface calculations that are usually based on floats. (google:41)
- Yes (although most compilers (get configured to) warn without a cast and that’s fine, but with the appropriate cast it must work). At least the comparison to `void*` or `char*` should always work since it must be possible to have programs/components deal with data in an opaque manner. Mixing function and non-function pointers might get you in trouble on certain weird architectures, but is usually (e.g. x86, ARM, ARM64) fine as well. (google:43)
- You’re assuming casting, right? Q is a bit confusing. (libc:1)
- Comparing incompatible pointers is a constraint violation I think.

But after converting to `void*` or some common type (that can represent both pointers) then the comparison is ok I think.

I’ve seen hash table implementations where the key was `void*` so pointers were compared internally and different types of pointers were inserted. (libc:3)
- Particularly common when there’s a lot of casting going on or weird tricks with unions. (libc:4)
- It shouldn’t be relied on, because the pointer is allowed to assume that two pointers to different type cannot alias.

I’ve seen code that relied on it when a struct was like

```
struct A {
    struct B b;
}
```

The code compared a pointer to `b` with a pointer to a ‘struct A’, knowing that they would be the same if `b` was contained within the struct. (libc:7)
- I’m assuming that one of the pointers are cast to `void*` or `char*`. (llvm:0)
- Comparisons of pointers to fundamental types like this break C strict aliasing rules. (llvm:2)
- We work with `void*` extensively and this is reliable. (llvm:3)
- Any sort of code that is providing a generic structure utility could end up doing this. Consider the following example:

```
are_my_structs_equal(void *s1, void *s2) {
    if (s1 == s2)
        return 1;
    // Figure out what kind of struct each pointer points at and compare
} (llvm:5)
```
- Strictly speaking, 6.5.9:2 in C99 only allows comparing `void*` to a pointer to an object type as heterogenous comparison. (regehr-blog:0)
- Don’t both pointers have to be to the same type or to `void` or am I thinking about argument passing? I generally avoid mixing pointers with different types. (regehr-blog:7)
- The "shouldn’t" applies to portable code bases. For a code base expected to run only on one architecture, I believe this is reasonably safe (assuming the architecture supports it). (regehr-blog:9)
- I have seen code that casts one pointer to the type of the other, then compare.
  (regehr-blog:15)
- exception: polymorphic types, where it is meaningful. (regehr-blog:16)
- Using pointer as identity. (regehr-blog:21)
- I usually cast them to void* and then to a pointer-sized integer type before
doing this, just to be on the safe side.

Sometimes we want to do those kind of comparisons in asserts, e.g. `assert( startPtr + currentSize == currentPtr )` or something along those lines. (regehr-blog:26)
- We treat all pointers and pointer-sized integers as exchangeable and comparable. (regehr-blog:28)
- There’s no requirement for pointers to different types to have the same structure. There could be a tag to the type in the pointer but in most machines a pointer is a pointer and comparisons just work. (x11:3)
- We recently had to deprecate/remove one of the linked list types from netbsd’s `<sys/queue.h>` because it stopped working with gcc 4.8 due to invalid comparisons of exactly this type.

There’s another one in there with similar problems that hasn’t broken yet.

This is wrong though and I would expect a program checker to complain about it.

See http://nxr.netbsd.org/xref/src/sys/sys/queue.h#666 (appropriate but coincidental line number) (x11:5)
- You can’t normally compare the actual pointer, but I would expect to be able
to compare the pointer after an appropriate cast. (i.e., if (int_p == (int *)struct_p) { ...} )

It’s not uncommon, particularly in systems programming, to have to cast pointers to different types. Particularly when dealing with device programming, I can imagine a situation where you might do something like the above. (xen:2)
- If I remember the spec correctly, a pointer to a structure and a pointer to
its first member of the structure are required by the C standard to have the same
value, and compare equal.

Other than that, depends entirely on -fno-strict-aliasing. (xen:3)

--------------------------------------------------------------------------
[7/15] Pointer comparison across different allocations
Can one do < comparison between pointers to separately allocated objects?

Will that work in normal C compilers?

<table>
<thead>
<tr>
<th></th>
<th>yes</th>
<th>only sometimes</th>
<th>no</th>
<th>don’t know</th>
<th>I don’t know what the question is asking</th>
<th>no response</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>191</td>
<td>52</td>
<td>31</td>
<td>38</td>
<td>3</td>
<td>8</td>
</tr>
</tbody>
</table>

Do you know of real code that relies on it?

<table>
<thead>
<tr>
<th></th>
<th>yes</th>
<th>yes, but it shouldn’t</th>
<th>no, but there might well be</th>
<th>no, that would be crazy</th>
<th>don’t know</th>
<th>no response</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>101</td>
<td>37</td>
<td>89</td>
<td>50</td>
<td>27</td>
<td>19</td>
</tr>
</tbody>
</table>

Comment

- less that / bigger than pointer has no meaning, unless within the same array
  (main2.2:3)
- it’s normal to compare pointers from different allocations to check if two handles to objects are equal for example (pointing to the same obj) (main2.2:5)
- E.g. in comparison callback for qsort() (main2.2:6)
- I’d expect this to be UB, so I think it may or may not work depending on many factors (main2.2:7)
Any sorting code! (main2.2:8)
- Again, programs are written to assume a flat address space.

I can think of actual situations where I would rely on this comparison behaviour, whereas I would usually avoid xor-linked-lists or pointer arithmetic between objects. For example, to get an arbitrary sort of allocated objects or to check if a pointer is part of an object. (main2.2:9)
- no, because it’s UB. (main2.2:10)
- Some memory allocators rely on it. (main2.2:17)
- Unless you’re writing code in a memory allocator, you have no business doing anything other than == or != comparison between pointers. (main2.2:18)
- again, there might be different address spaces (main2:0)
- Same as above -- having pointers ordered is probably important for tree structures, and I assume it’ll work, maybe with casting to void* or integer. (main2:8)
- Just doing sorting for, eg, associative containers. Only need is that result is consistent. (main2:9)
- Happens to work most of the time. (main2:10)
- Useful for example for some implementations of dictionnaries that require all keys to be comparable. (main2:11)
- std::map with pointer as key! (main2:12)
- I’m sure such code exists, but that would be insane with a "normal" runtime. (main2:16)
- I’ve certainly seen people do hash tables keyed on structure addresses as an alternative to adding more fields to the structure. (main2:19)
- UB in C++, likely that C compilers differ in how they treat it. (main2:26)
- have seen this for sorted collections of pointers (sorting based on pointer numeric value, not on target value - it is not a "meaningful" order but it is at least assumed to be possible, if we want to iterate over the collection in the same order each time, say) (main2:28)
- some non-built-in alloca code (main2:29)
- K&R says < and > is only ok within the same array or allocation, but it likely would work across allocations just not be very useful.

I’ve seen "for (p=Start; p<End; p++) {do stuff}" as a fairly common idiom. (main2:30)
- Again, this doesn’t work well if pointers are implemented as base+offset. (main2:32)
- don’t second guess the memory allocator (main2:34)
- Garbage collectors can do this. (main2:35)
- [Again the restriction in the standard is there only to satisfy 8086] (main2:37)
- 50/50 chance it’ll be right. But only by luck. (main2:40)
- How exactly would you make an std::map indexed by pointers if you couldn’t? (main2:50)
- This is needed for building BST-based sets of heap-allocated objects. (main2:54)
- I’m guessing that you mean using < to see which one was allocated first. I believe you can do < comparisons but the answer is generally useless. (main2:55)
- This happens when people make C++ "std::map"s of pointers, as a method for attaching additional data to objects whose layout is fixed. As I recall, std::less is specialized so that this will work, but honestly I don’t see why the standard wouldn’t just allow < to work across different allocations. What is do be gained by doing this? (main2:59)
- Not the comparison but quite famous example of clang optimization:
  int *arrayA; // on x86, not aligned, say, the address ends with 0x2.
  int *arrayB; // on x86, not aligned, say, the address ends with 0x0.
  int magic = ((char*)elementB - (char*)elementA) << 1;
  results in an unexpected assumption by the compiler that elementB and elementA should have a sizeof(int) divisible pointer difference, so it generates
  int broken_magic = (elementB - elementA) >> 1 instead. (main2:64)
- This typically works on flat-addressing systems and typically fails on segmented memory systems (because it tends to only compare the offset). (main2:69)
- It’s undefined behavior, but library implementations may rely on it, I don’t know. If so, I suspect it’s fairly safe to do so, but why risk it? Not sure wha
For example, the common implementation of std::less in C++ relies on this. (main2:70)
- It just calls \(<\) on the pointers and relies on the implementation-defined property that \(<\) is a total ordering.) The alternative implementation suggests I’ve heard is to cast the pointers to intptr_t oruintptr_t. (main2:74)
- Same comments as the previous one. (main2:80)
- Example: A check for aliasing of buffers (main2:87)
- Again, what does ‘separate allocation’ mean?

But even in malloc() case, yes, breaking this pattern would break a lot of code -- e.g., many data structures that uses pointer identity for elements. Even, say, typedef int *HandleType; std::set<HandleType> in C++ (I realize that the question is about C, but "normal C compilers" aren’t usually expected to change semantics between C and C++, many programmers don’t even realize there isn’t a subset relationship). (main2:90)
- same as question 4 really (main2:91)
- you can do it but the results are unspecified. So you can’t make any reliable inference from it. (main2:92)
- More strict aliasing problems. (main2:94)
- C++ std::less does something like that. Wouldn’t surprise me if others do that too. (main2:96)
- Again I don’t know what compilers will do in practice, the spec does permit miscompilation.

Also as above it’s entirely likely that an allocator implementation will want to do this. (main2:100)
- A pointer is a comparable value. For example, if you have a binary search tree of pointers, you rely on this.
- Is this asking if you can do intptr_t(pA) < intptr_t(pB)? ... In that case the answer is definitely a ‘yes’ to both...
- Why the hell would anyone do this? (main2:121)
- Some hash table implementations keep ordered lists of pointers of some form.
- See my comment on 3/15. This is undefined behavior, but I’ve never seen a compiler smart enough to know when it happens in real world code. (main2:126)
- Implementing memmove is no fun without that. (main2:130)
- embedded with constraint on malloc: malloc used only during init, never freed (main2:133)
- Any implementation of binary trees for pointer types that I know of will do this (C++ even went as far as giving a special exception for std::less). (main2:137)
- It will return true or false, which one it return is a potluck. (main2:144)
- Assuming uniform memory. Comparing addresses may be idiomatic for an allocator implementation. (main2:147)
- heap allocator trees (main2:148)
- Oh probably none of this is legal. Doesn’t matter. "Shouldn’t" here is like someone telling you you shouldn’t jaywalk. (main2:156)
- I wrote some code that does sorting of linked list items by pointer during traversal to improve cache performance that relies on this. (main2:163)
- How is this question different from Question 4? (main2:165)
- It does not make sense in a high level application (it should not depend on where the memory is allocated)

Only low level code might use it, for special purposes. (main2:173)
- std::set< void*> (main2:176)
- I am thinking of cases like sets and trees and such keyed by pointer addresses, but which use a pointer rather than intptr_t. (freebsd:1)
- My initial answer was "no" but memmove() needs to identify if the source and destination overlap -- even though they may be separately allocated objects. (freebsd:6)
- It does not seem commonly known that comparison between pointers to different objects is undefined behavior. (freebsd:8)
- It probably only works reliably when the compiler cannot 'see' whether the pointers point to separate objects, e.g. in the case of extern calls (with no LTO) providing a compiler barrier.
Code that wants to rely on this should probably cast to uintptr_t to make the comparison, but that imposes assumptions on the definition of this implementation-defined conversion.

(gcc:2)
- It’s necessary if you’re going to use pointers as keys to a binary tree. (gcc:3)
- Comparison may only work if cast to char* or void* (google:2)
- The standard says the behavior is undefined, but as far as I know (or in practice), there is a strict ordering (regardless if that makes sense), you can use a pointer type as a map<> or set<> key (which I think runs from a standard less / operator <). Regardless, a preferred ‘access by ptr value’ for such would be to use a hash / unordered_map (google:4)
- < comparison doesn’t make sense if they’re not of the same type (google:5)
- map<pointer, extra data> would rely on that. (google:7)
- Pointers can be used as keys in a map. (google:11)
- Imagine you want to maintain as a data structure a set of widgets (of type struct Widget* for example), with operations
  - add a widget to the set if it is not already there.
  - list all widgets
  - delete widget if present

If you assume you have an order on pointers, you can use a binary search tree. Otherwise, you cannot.

(you can also cast them to integers and hash them, or something, but I suppose you also think this is an invalid use of pointers)

(google:13)
- Very useful for sorting, e.g. to eliminate dupes in an array of pointers. (google:14)
- This seems to work because in C++ it is common to use a pointer as the key type in a std::map, which by default uses the < operator as a comparator. (google:18)
- Any time you have map<type*, type2> in C++ code, you are implicitly performing these comparisons. (google:21)
- Again, pointer representation may vary across architectures. (google:24)
- hash-like structs (google:27)
- Work, in the sense that it won’t throw an error. I’m not sure the result is sensible*
(google:29)
- A legit example is a map from pointer to something. (google:30)
- I don’t believe it is portable, and I think it’s undefined behavior, so you shouldn’t rely on it (for a practical reason a system might not support this: systems with segmented memory might just compare offsets).

Now having said that I do know of code which uses this provide ordering for a user-defined type which doesn’t have to dereference the pointers.
(google:34)
- I imagine this could be used internally to a memory allocation system, to track allocation blocks or similar. (google:35)
- If the objects are in different address spaces, the compiler may or may not include an address space identifier (e.g. segment id) in the pointer, and may or may not compare this along with the pointer. (google:36)

I am aware of code that uses pointer comparisons to check whether a pointer is within an object. Even this is not defined, although it will "usually" work. Some Quake engines e.g. use this approach to verify pointers derived by QuakeC VM code are within the VM’s memory area.

On platforms with sizeof(void*) == sizeof(ptrdiff_t), it will "usually" work assuming the object size divides the difference. Wouldn’t count on this though. (google:36)
- You might do this for an arbitrary (but stable) sort order. (google:38)
- Is this different from question 3? I know it’s different from question 6. (google:39)
- I would expect this to return whether one pointer has a lower runtime value than another, but I wouldn’t expect that to be meaningful. malloc could return a n address from any part of the address space. (google:40)
- AFAIK, a pointer is just a value and it supports comparison. Comparing point ers to separately allocated objects is meaningless though. (google:41)
- Um, duh. This is for using pointers as keys into a map. (google:42)
- Occasionally, detecting where (stack, heap, bss, etc.) a certain object is a llocated is useful and can be done this way. (google:43)
- ordered data structures often use pointers as keys. It typically works, but in some memory models it might well fail. (google:46)
- sort ordering objects in data structures (libc:0)
- Again, it’s not guaranteed by the standard, but lots of code does it, e.g., for binary search using the pointer as a key. (libc:1)
- same comment as for [3/15] (libc:3)
- In practice, you can do this operation, but the results are basically undefi ned. I know of code that uses it by accident, but it’s always a bug. (libc:4)
- Usually memmove relies on this. (libc:6)
- I’m pretty sure it would "work", but I don’t really know why you would do th at. (libc:7)
- C++ at least says that if you do this, you don’t get a stable answer. It doe s, however, provide std::less, which will give you a stable answer. Containers l ike std::map rely on the ordering of pointer values, so there are real programs out there that rely on this. (llvm:2)
- Occasionally there might be some use to doing that in a debugging environmen t. It also might be found when using pointers to hold integers, that usage is n ot unheard of for us. (llvm:3)
- sorting list of pointers, putting pointers into balanced trees, ... (llvm:4)
- memmove() again. (llvm:7)
- The example that I know of is p <= q where q points to a char object and p i s a null pointer, in a condition in a loop where p starts to point inside the sa me array as q after the first iteration, in the QuickLZ library. This is discus ed further at http://stackoverflow.com/questions/7058176/on-a-platform-where-nul l-is-represented-as-0-has-a-compiler-ever-generated-unex (regehr-blog:0)
- It’s certainly UB though. (regehr-blog:4)
- The comparison can be performed, but the result won’t be useful unless you p lan on doing some of the other crazy stuff this survey is asking about. (regehr- blog:7)
- seen it used for stable lock ordering: if you need to lock two different "x" simultaneously, always lock object at lower (or higher) address first. (regehr-blog:11)
- Any sort of map data structure using pointers as keys could hit this. I cou ld see this being useful in rare circumstances (e.g., to store data associated w ith opaque pointers returned from a third-party library). (regehr-blog:12)
- I don’t work with dynamically allocated memory, so no relevant experience. ( regehr-blog:14)
- Must be total ordering, actually. (regehr-blog:16)
- Clearly this is undefined behaviour, but I’ve certainly seen code which reli es on this. The best example I can think of is the GNU implementation of alloc a, which did this to determine the direction of stack growth. (regehr-blog:20)
- store pointer to structure in a set based on balanced tree and using pointer directly in comparation function (regehr-blog:21)
- I can not see what purpose doing so would be. (regehr-blog:22)
- Using addresses as keys in a sorted data structure (binary tree, etc.) is on e case that might come up in practice.

For safety voodoo reasons, I would be tempted to cast them to a pointer-sized in teger type first. Or else read carefully through the standard. (regehr-blog:26)
- Efficient versions of memmove. (regehr-blog:30)
- Most mallocs need to do that internally to store an address indexed list. (x 11:3)
- I don’t write code that does this (because I learned C on MS-DOS where it di dn’t work) but many/most people do. Things I’ve seen random pointer comparisons
used for: inserting into trees or tables; establishing a uniform locking order.

Also, memory management code tends to need such logic; e.g. any malloc that main
tains any kind of sorted or indexed structure of memory blocks. But this gets ba
ck to what an allocation is in that context. (x11:5)
- but with what result? (x11:6)
(xen:3)

[8/15] Pointer values after lifetime end
Can you inspect (e.g. by comparing with ==) the value of a pointer to an object
after the object itself has been free’d or its scope has ended?

Will that work in normal C compilers?
yes: 209 (66%)
only sometimes: 52 (16%)
no: 30 (9%)
don’t know: 23 (7%)
I don’t know what the question is asking: 1 (0%)
no response: 8

Do you know of real code that relies on it?
yes: 43 (14%)
yes, but it shouldn’t: 55 (18%)
no, but there might well be: 102 (33%)
no, that would be crazy: 86 (28%)
don’t know: 18 (5%)
no response: 19

Comment
- A diagnostic or security function which checks for double-free might well do
this. (main2.2:0)
- The pointer is just an integer value, only dereferencing is not possible any
more (main2.2:3)
- you may want to check if you have just free’d a specific object you know abo
ut through a pointer. (main2.2:5)
- I think this is UB as well (main2.2:7)
- Pointers used as opaque references. (main2.2:8)
- Depends what you mean by ’can you’. You _can_, in that I would expect a com
piler to generate code to check the pointer rather than doing something random.

But in either case I have no clue when or if the pointer might be reused by the
memory for another object. Code that does this is most likely broken.  
(main2.2:9)
- AFAIK it’s undefined behavior to even read a pointer to a deallocated object
. (of course dereferencing and reading the object itself is an even bigger no-no)
.
(main2.2:10)
- It seems to me to be legitimate to use the values of pointers even when what
they are pointing to is invalid, as long as you don’t dereference them. I have
used pointers as keys to STL map, but maybe that’s out of scope here ... (main2.
2:18)
- I can imagine some architectures where this would break, but this works fine
on modern Unix systems and modern CPUs with a flat address space. But who knows,
this may completely break if one uses a pointer-checking mechanism (preventing
use-after-free, access out of range, etc.) since it may (rightfully!) trigger t
hat mechanism. (main2:8)
- Can’t see why I’d do this. (main2:9)
- That still happens to work. (main2:10)
- But because of risk of address reuse rather than optimisations (main2:12)
- See the realloc implementation in the malloc in FreeBSD’s rtld-elf. I still
don’t understand what they are trying to do, but there is even manpage documenta
tion in old unixes about the bizarre behavior they are implementing when you pas
s a free()d pointer to realloc. (main2:16)
- I would expect this to be safer with malloc/free than pointers across scopes
where the compiler has (possibly) more information about object lifetime. (main2:27)
- I have seen code that tried to detect double-frees by remembering the values of
  pointers that had already been freed. (main2:28)
- The pointer itself is still valid, and can be compared. Dereferencing the po-
  inter can't. (main2:29)
- Microsoft PREfast has a warning for use of a pointer after freeing it. (main
  2:30)
- I expect this will usually work, but might fail when run on a capability arc
  hitecture. (main2:32)
- I'm pretty sure garbage collectors may do this. (main2:35)
- "or its scope has ended" - is this a complex C++ question?

You might need to define more clearly what you are asking. A pointer is a value
which does not cease to have a value because you happened to pass that value to
a function called free (or any other function annotated with _Frees_ptr_) but t
he set of things that it would be reasonable to do with such a pointer would be
extremely limited. (main2:37)
- Again the option I'm looking for is "I'd hope so". A reasonable use would be
to update old pointers after having reallocated / moved stuff around. (I think
a vaguely similar thing is done in Perl when creating an ithread: relevant resou-
rces are cloned, a table is constructed mapping old pointers to new pointers and
appropriate substitutions are done on all pointers. It doesn't quite fit the ca
se of the question since the old objects still exist, but it's easy to imagine a
similar situation but with reallocation/movement rather than duplication.) (mai
n2:50)
- Again, while I understand this may be undefined behavior, I can't imagine wh
at optimization would be gained by blocking this. Where I've seen this is code
like this:

```c
free(myptr);
release_extra_data_keyed_by_pointer(myptr); (main2:59)
```
- For instance, optimizers are likely to assume that the results of two differ-
  ent malloc calls are different, even if there's an intervening call to free. (ma
in2:69)
- It's undefined behavior to access a pointer after the object it pointed to’s
  lifetime has expired (such as after free) (main2:70)
- IIRC, the standard is vicious here; once a pointer is freed, *all copies* of
  that pointer are sent to an undefined state, nearly the same as being uninitial-
  ized except that you are guaranteed a stable value each time you observe it.

A common pattern that relies on this is calling realloc and "checking whether it
moved" to decide whether to update other copies of the pointer. (main2:74)
- After all, it's just an integer (main2:75)
- The pointer is just an address. The pointed content is not accesible, but th
  e pointer can be compared without any problem. (main2:78)
- Dereferencing a dangling pointer is definitely an undefined behavior. What i
  s less well-known is that using the value of a dangling pointer itself is also u
  ndefined behavior. (main2:80)
- If you're asking whether the you can still inspect the pointer value, then y
  es, you can (though have fun with that). If you're asking whether the value is v
  alid, then it may be, but it may not be.

(main2:86)
- ... there might be such code, but it would be totally crazy :) (main2:90)
- Undefined behaviour. (main2:94)
- It works most of the time, but malloc() can return the same value twice if t
  he first one is free()d. (main2:96)
- But it's hacky debug code that doesn't ship, thankfully (main2:98)
- Pointer values become indeterminate after object lifetime ends, presumably f
  reeing the compiler to stop tracking the value of the pointer. I assume that com
  pilers will take advantage of this but have not checked.

This is bonkers, since programmers who care about that optimization opportunity
are free to not inspect the value of such pointers. (main2:100)
Great source of errors! (main2:102)
- Just learned this one yesterday. (main2:106)
- Once a pointer has been passed to free, it should no longer be referenced
  in any way. Naturally, the local variable doesn’t change, but comparing a freed
  pointer with another is useless. The pointed to location might be reused at any
time. (main2:115)
- Use after free: not even once (main2:121)
- I have seen plenty of bugs where code does this and then the address is reus-
ed by the allocator later and bad things happen. (main2:123)
- Common bug. Since comparison is usually just address comparison no error is
  raised by comparing pointer that is out of scope. (main2:127)
- Spec says it’s undefined, but in practice I’ve never seen anything but the o-
bvious behavior. (main2:129)
- Might have used that for sanity checking at some point (main2:130)
- no free allowed => not possible (main2:133)
- I guess someone may have two objects, does not know if they are alias and ha-
s to free both.
  In that case, he might free one, compare pointers and if they were not equal, fr-
e the other. (main2:144)
- Assuming you mean the former address of the expired object.
  It’s an error. It’s not valid to compare a former (invalid) address to a living
  (valid) one.

It’s giving pointers a semantics which isn’t reflected by their values. Multiple
objects are implied by a region allocation. Two pointers which each happen to re-er to a former object each with the same lifetime (valid to compare ==) aren’t
distinguishable from any other pointers. (main2:147)
- pointers are a variable, just like an int. what they point to is irrelevant
  for pointer comparison. (main2:148)
- Come on, I know of _memory-diagnostic_ tools that rely on it. (main2:156)
- A pointer in your program is not destroyed or overwritten by calling free() on
  it, or when the object it points to has gone out of scope. However, the objec-
t can’t assumed to be valid after either of these things has occurred, and the m-
emory may have been recycled, or it may not have been recycled (yet). So this is
not a safe practice in any way. (main2:158)
- I’ve seen assertion macros that check the contents of pointers to check whet-
er we are using freed pointers (the msvc CDCDCDCD, EDEDEDED, FDFDFDFD pointers).

I’ve also seen a pooling mechanism for a sound system that allowedsounds to be
accessed via what would conventionally be freed pointers (using a rolling ID tha-
t is compared against). (main2:163)
- So long as you do not actually dereference the pointer, I don’t see the prob-
lem in inspecting its value. I can imagine a use case in which you might wish t
o check wh I can imagine a use case in which you might wish to check wh (main2:
165)
- Once an object has been deleted, using any kind of reference to it is a bug,
in my opinion.
I can’t think of any use of such a test after an object has been deleted (such a
test is only useful before deletion). (main2:173)
- You can, for instance, check whether it was NULL. (main2:176)
- Use-after-free bugs are unfortunately common. ‘==’ inspection would be a va-
rient of that.
(freebsd:6)
- This seems related to use-after-free issues. I believe that compilers will
let it through, even though it’s not something you should do. (freebsd:8)
- The question seems vague to me. I am interpreting this question as meaning
"will code that uses == on a pointer to a deallocated object crash?", for which
I think the answer is that, for a normal compiler, it will not crash (though of
course it is undefined behaviour). I can’t make any predictions about what the
result of that comparison will be. (gcc:1)
- Relying on what? The pointer still exists but what it points to is meaningless.
For example, you can open a file with:
```c
FILE *f = fopen...
<do some work that may jump to label:>
```
fclose (f);
f = NULL;

label:
if (f)
  flose (f);

here we compare f even though the FILE it was pointing to was closed. I think that is valid. (gcc:4)
  - As long use the memory is still readable by the process then this will return some data, potentially undefined parts of the stack, the original non-overwritten data, or new data rewritten into that allocation of memory. (google:1)
  - I think there is nothing wrong with code that does not de-reference the data but does other stuff, say naive code sample:

    delete p;
    if (p != nullptr) {
      ++objects_deleted_;
    }

    (google:4)
    - A hash table stored extra data for objects, indexed by their address. Sometimes when the data was freed, it was first freed, then the hash table entry was deleted. (google:7)
    - This would generally work (and I can hypothetically imagine scenarios where someone might want to try, but they shouldn’t) but some analysis tools will explicitly scribble on the pointer to make sure you don’t do it (because it’s a terrible idea). (google:11)
    - I say it shouldn’t because keeping dangling pointers around is scary business. But this will happen in the case of linear search over a collection of maybe-valid pointers, which I have seen. (google:21)
    - You can’t tell whether something new has been been reallocated into that memory. (google:27)
    - A good compiler should show a warning but I don’t know if anyone has implemented it (google:30)
    - I’m sure this will work on many C compilers. What I don’t know is if there are compilers where it won’t work, although I can imagine that being the case. (google:32)
    - I don’t think you can do this if the value is the pointer type itself, but if you converted to a string or integer the compiler of course can’t stop you. Now the address could very well be reused at any time so the utility is questionable. (google:34)
    - Keeping a pointer around after the thing it points to has been freed is symptomatic of something wrong with the program design. (google:36)
    - I’ve seen cyclic linked list implementations do this to check whether they just freed the last remaining list item. (google:37)
    - Objects going out of scope do not invalidate access to their pointers. That’s the whole dangling-pointer problem that C has. (google:38)
    - I would expect the pointer value to be the same after the object it points to is freed. This might be used, e.g., to remove a pointer from a collection. I don’t have any specific example though. (google:40)
    - As long as you don’t dereference the pointer, you can use it. I can’t think of a legitimate use, but you can. (google:41)
    - This is the result of a stupid decision by the standards cmte. a free’d pointer cannot be compared. (google:42)
    - The pointer should still have a defined, stable value, but otherwise this sounds pretty useless. I guess it could be used to check if the pointer used to be NULL before, or which area (stack, heap, etc.) it was allocated in. (google:43)
    - Freestanding code (c runtime implementation or os kernel) may make such assumptions in some cases. (eg the c runtime implements free so it has full control over free’d pointers or munmap’d objects, but it should not inspect pointers to automatic variables that went out of scope).

    i think this works in practice and there might be code that relies on it, but it shouldn’t.
Most compilers aren’t going to use a trap representation or otherwise intentionally break this, but it’s seriously dodgy and the compiler should scream about it if it can detect it, since it’s a great way to create bugs like use after free. (libc:4)

- C is pretty clear that you can’t look at the bytes of a pointer to an object that has been free’d. (llvm:2)
- The value of a ptr is not changed by freeing what it points at. It can be useful to know a ptr of a freed object for leak detection. (llvm:3)
- You can’t dereference the pointer, but the value remains valid. The only good use for it I can think of is to log a debugging message (which would only be useful if one also logged the allocate).

In fact, I have logged such messages myself when unloading a loadable kernel driver (because all evidence of what had been at those pages was gone; so, anything faulting referencing the unloaded driver would be a complete mystery). (llvm:5)
- As discussed in Q4, the current stable version of ntpd does this. (regehr-blog:0)
- The pointer’s value will no longer reference valid memory, so the pointer cannot be dereferenced. (regehr-blog:7)
- Specifically, I have seen code that relies on comparing the "this" value after calling delete. (regehr-blog:9)
- I think this would work, but I am not motivated enough to test it or scour the spec. (regehr-blog:12)
- as i said pointer in flat memory model is just an index (regehr-blog:21)
- At a guess the pointer will still contain the old address, it should be safe to do assuming it is not dereferenced, unless the spec says otherwise. (regehr-blog:22)
- As I mentioned above, this is a bad idea. On certain compilers, it won’t work. Example: http://trust-in-soft.com/dangling-pointer-indeterminate/

- I’ve seen this done, but it was years ago... code hangs around for a long time however. (regehr-blog:26)
- Cleanup code (x11:0)
- If you are evaluating x == y, where *x has been freed and y points to a newly allocated object, then in very rare circumstances you could get a false positive. (x11:1)
- maybe it might depend on the deallocation. If it’s on the stack in the same function, it probably hasn’t been overwritten unless your compiler is really aggressive about reusing/reclaiming stack space. (x11:3)
- A normal (i.e. not linting) compiler won’t null out or otherwise mangle references after they’ve been passed to free, so code that does this will generally work. I’d expect that in order to break one would have to have a loop such that the compiler can derive some useful information by assuming a free call hasn’t been reached yet because the pointer value freed in that call is being used... such that this information is actually wrong. But I’d expect such code to be very rare.

A bigger issue is what happens if you have (as bugs) dangling references in a data structure or whatever after a free call. Obviously using dangling references is unsafe; but also, if a compiler is able to conclude that they’re dangling when used later, it ought to complain rather than make silly assumptions and generate wrong code. (x11:5)
- freeing the object before removing its pointer from a container (x11:6)
- I guess that is one of the standard issues you would find with more complex static analysis methods such as coverity (pointer use after free). (xen:1)
- free() is not permitted to alter the pointer passed, which means the pointer shall still have its previous numeric value.

Whether this subsequently compares equal or not depends on how your compiler is feeling, but I seem to recall that it is UB to do so. (xen:3)
Can you (transiently) construct an out-of-bounds pointer value (e.g. before the beginning of an array, or more than one-past its end) by pointer arithmetic, so long as later arithmetic makes it in-bounds before it is used to access memory?

Will that work in normal C compilers?

- yes : 230 (73%)
- only sometimes : 43 (13%)
- no : 13 (4%)
- don’t know : 27 (8%)
- I don’t know what the question is asking : 2 (0%)
- no response : 8

Do you know of real code that relies on it?

- yes : 101 (33%)
- yes, but it shouldn’t : 50 (16%)
- no, but there might well be : 123 (40%)
- no, that would be crazy : 18 (5%)
- don’t know : 14 (4%)
- no response : 17

Comment

- pointer arithmetic is just normal arithmetic (main2.2:3)
- Often seeing a code which uses pointer to 1st array member and pointer to th e after-last array entry. Last member is then (end - 1). (main2.2:6)
- That sounds like UB to me as well, so I’d assume it might work for certain c ases (main2.2:7)
- For pointers to variable-sized or unsized objects, various computations are often made, such as allocation code that puts a variable-sized array behind a st ruct, and constructs pointers to entries of that array. (main2.2:8)
- More useful than the other pointer arithmetic question, but I am suspecting that this is still probably undefined behaviour? (main2.2:9)
- It’s technically undefined behavior APAlK, but I’ve never actually tried it. I guess some compilers may use this instance of UB to perform clever optimizati ons, but I don’t know what’s the state of the art in this aspect. (main2.2:10)
- One-past is ok? But two-past is not? Would expect code that constructs a poi nter, then checks if it’s within a valid range before using it. (main2.2:15)
- There is no thought crime in C ... you’re only in trouble if you dereference an invalid pointer. (main2.2:18)
- this was mentioned in the (I think) CCured paper (main2:0)
- What crazy architecture/compiler would that not work on?

A non-dereferenced pointer is just a number! Right? ......right? (main2:7)

- The Numerical Recipes in C rely on it; that’s widely-used code in the physic s community with some pretty horrible (and probably illegal) C code. This code e xplicitly stores and passes out-of-bounds pointers.

If I index a multi-dimensional array manually, then I there’s a chain of arithme tic like "p + i*di + j*dj + k*dk" or so, where p is a pointer and the others are integers, and I don’t pay attention to the order in which these are evaluated. This just may temporarily lead to out-of-bounds pointers, depending on the order of evaluation. (main2:8)
- Don’t think I do more than the OK 1-past end (main2:9)
- Also happens to work, but only one past the end is valid. (main2:10)
- I believe this is undefined, but don’t know if any compiler would break it. (main2:11)
- Tcpdump does a bit of this where they create a variable from an array and th en check it is in bounds (main2:16)
- Assuming it’s possible to get to a valid region via pointer arithmetic. (mai n2:26)
- Chances are this was tightened up and my answers are now wrong.... (main2:29 )
- Yes, but be careful about wrap-around. Also looks like a security bug, eg ca llloc(x, y) where x*y wraps around causing a tiny allocation to happen, then usin g the larger assumed result. (main2:30)
- The 8086 is gone. A pointer is a value. (main2:37)
- Possible, but it’s likely a bug waiting to happen. (main2:40)
- I’m pretty sure I’ve had such cases with TI not-too-bright compiler for their C6000 DSPs, where pre-decrementing an index before the start of a loop and pre-incrementing it before use resulted in the best code output. (main2:50)
- This may be undefined behavior, but I’ve written such code myself and it would be a pain to try and fix it. Sometimes such code becomes incredibly ugly when trying to fix it. (main2:59)
- Yes, I’ve seen string classes that allocate sizeof(int)+strlen bytes, store the char * pointer to the malloc result+sizeof(int), then make use of the fact that the string object’s representation just looks like a char * so you can pass it to printf without calling a .c_str() on platforms where it reads the bytes of the object to pass it through "...". It then used the int as a reference count, and the destructor did the decrement and possible free. (main2:66)
- Does this count?
- container trickery that uses ((void*) 0)->someMember as a way to do "offsetof" (main2:67)
- Use case: arrays with a lower bound not equal to zero (like in Fortran)

```c
int lbound=10, ubound=20;
const int asize = ubound - lbound + 1;
double *arr = malloc(asize * sizeof(double));
double *a = arr - lbound;
for (int i = lbound; i <= ubound; i++) {
    a[i] = 0.0;
}
```
- Some sanitizers check for this. (main2:69)
- Though this is undefined, I figure most compilers may permit it, but you may well accidentally overflow a register depending on the underlying representation of the pointer and how near it is to the limit, etc. (main2:70)
- Yeah, we didn’t even bother with this one in clang -fsanitize=undefined. (main2:74)
- Undefined (main2:80)
- Some f2c (Fortran2C) converter produce such code (main2:87)
- This should definitely work. The mental model of pointers to certain type T being sufficiently wide integers of a distinct type for every distinct T is very deeply ingrained in engineers’ minds. I just don’t know how it would be possible to change it, or to even explain pointers in a different way to a novice. (main2:90)
- Pretty sure this one I’ve seen buggy code optimised away by real compilers. (main2:92)
- Undefined behaviour. Also a good source of security vulnerabilities. (main2:94)
- I assume that compilers will miscompile it, but I’ve not checked. (main2:100)
- I know that the standard doesn’t allow it, and as400 will break, but there is plenty of post-decrement pointer code that works fine on most flat address machines. (main2:111)
- Stack frame manipulation is "fun", pretty common in malware (main2:112)
- Not even intermediate values should point outside of an array (except just past the end). (main2:115)
- I don’t see what’s wrong with this (main2:121)
- I think this is legal as long as you don’t read the value. (main2:123)
- Common to get address of array element "-1" before passing to code which pre-increments pointer to element "0" (main2:127)
- I guess it depends how much "how of bounds you go."
- Out if array limit is probably fine. Wrapping 2^32 or 2^64 could be a problem. (main2:144)
- Pointers don’t have semantics. So typeof("out-of-bounds" pointer)==typeof(pointer). "Can you make a pointer then later make it a pointer before it is used?"

With taint (poison) it’s interesting as depending how it is created, an "out-of-bounds" reference should be tainted. And taint can’t be removed by simple operations. With taint it could be idiomatic to iterate until exception. But that isn’t...
Thinking of buffer overflows it would be an advantage to invalidate references before data could be written past the buffer.

Usually, an exception is generated when an invalid reference is dereferenced, not merely by creating one (back to the problem that pointers don’t have semantics – how do you know if a value in a register is a pointer or not?). (main2:147)
- null, EOF markers, etc. (main2:148)
- All the time. All the time. (main2:156)
- I know that the standard says you may construct a pointer that points to one element past the last value of an array, but that it is not a defined operation to dereference that value. If you walk off the array in the other direction (forming the address prior to the zeroth element) you are creating undefined behaviour; you may create an invalid address. I think on most architectures this will not crash, but technically it should not be relied upon. (main2:158)
- I’ve seen hash tables that do this.

Also, in our industry it’s not uncommon to purposefully access off the end of the array. Eg. We do string operations a word at a time even though strings don’t have to be a multiple of 4 bytes in length. (main2:163)
- I’m pretty sure I’ve seen this before somewhere. E.g.,

```c
char* a = (char *)malloc(47);
char* b = a − 1; // Out of bounds.
b++; // Now OK.
b[0] = ’a’; // Should be fine. (main2:165)
- To make a vector/matrix start artificially at 1 (instead of 0), for example. This trick is also typically used when using a table as a function taking one integer argument having a strictly positive range, without having to modify the input directly (or for interpolation). (main2:173)
- for (char *p = A; a < p+n; p++) ;
```

I can construct it, the question doesn’t say I have have to use it.

The code above could actually fail if A’s last byte is at 0xFFFFFFFF (32bit), but I think OSs don’t give you memory there (as it doesn’t give you memory at NULL). (main2:176)
- This should be allowed. malloc is often used w/ these, and the compiler can not/should not know it’s bounds. (freebsd:5)
- I think current compilers will let that work, but haven’t tested it myself. (freebsd:8)
- This happens to be particularly dangerous when the address happens to be close to the top or bottom of address space.
  (gcc:2)
- Seems like that should be legal (gcc:4)
- I believe the danger is the overflow case, not the fact if it points to valid memory. The overflow / underflow case is undefined, for example, the below I believe is undefined:

```c
char *p = nullptr;
char *p1 = p - 1;
char *p2 = p1 + 1;
// Undefined what p1 + p2 point to (google:4)
- Some code happened to compute the final address of some data, in a way that was split across several lines, and the intermediate result was out of bounds. (google:7)
- I’ve seen pointers to before the beginning of an array in preparation for it iterating over it so that you can increment the pointer at the beginning of every loop.
```

I see no reason why this should fail on any modern system since the failure case for this would be low pointers in zero page where C programs aren’t allowed to mess around anyway.

It’s probably not a good idea regardless. (google:11)
- Tagged values. (google:12)
- This SHOULD work. (google:13)
- Non zero-based arrays "work" this way. (google:14)
- Format dependent indexing into objects. Some object representations keep the excess bytes of object after rounding to the allocation size in a header, and may transiently have a pointer beyond the object before subtracting them out. (google:16)
- Depending on the semantics of the question, I’ve seen this relied on for old WIN32 API code. (google:20)
- A pointer is still just an integer. I recall in the past being faced with this problem and electing not to allow an invalid pointer to be passed around. (google:21)
- Undefined behavior. (google:22)
- Useful for clamping array indices. Instead of clamping the index, you just allocate a way bigger array and start the zero index in the middle of it. The out of bounds, indices are then filled with clamped values. (google:23)
- This will only work if pointers are just dolled up integers. (google:24)
- Everyone writes ‘p+b−a’ instead of ‘p+(b−a)’ (google:30)
- Doing this sounds wrong, but for all I know it works, even possibly all of the time. (google:32)
- This isn’t portable because it invokes undefined behavior, but lots of legacy code does this, or at least I have memories of finding it in the past. It’s not legal to construct such pointers, but compilers are generally forgiving of this (as compared to actual dereferencing of them which would frequently be caught). (google:34)
- It’s only valid to point to locations inside an object or one past its end. On some exotic platforms it may not be possible to represent pointers anywhere else; but most real-world compilers use flat address spaces and don’t care. (google:36)
- I’ve seen such code but don’t remember where.

This, again, may obviously break due to overflows on platforms where size_t storage is smaller than pointer storage. (google:37)
- I’ve done this for things like bounds checking by overallocating the block and writing canaries at the ends. (google:38)
- The fact that you can is behind all the off-by-one errors and unlimited reads that can be exploited by attackers. (google:41)
- This would be useful for for implementing a min/max heap. (google:42)
- Feels like this should be fine but I can’t think of a relevant use case. (google:43)
- Compiler might optimize out UB. (google:46)
- see [3/15] about how the compiler can use this for range-analysis.

This is unfortunately easy to do by mistake so i’m sure it’s common.

it allows the construction of pointers with difference that cannot be represented as signed ptrdiff_t which is problematic.

(and most likely it makes c implementation expensive where pointer overflow is not valid) (libc:3)
- This simplifies some offset pointer calculations, and it can be hard to keep all intermediate results within bounds. (libc:4)
- I know it won’t work on either hppa or ia64 (ilp32), due to how segmentation operates. But those are pretty obscure these days.

I know that plenty of compilers optimize termination of loops based on end-of-array conditions, so there are likely to be all sorts of issues there.

It probably will work if, while out-of-bounds, the value has been cast to an integral value, and the cast back to pointer happens after the "later arithmetic". (libc:6)
- Barring integer overflow, yes, this pretty much works just fine. (llvm:2)
- This may be commonplace to create a sentry. You can put any number you want into a ptr. (llvm:3)
- There is pretty much no such thing as an out of bounds pointer in C. They can always be cast as an integer type and back again.

Consider a debugger that is operating on memory in a different address space. The debugger can (and at least one I know of does) do address computations based in a different address space. Load and stores have to be via access functions, but address computation does not. (llvm:5)
- Any intermediate states that can be optimized away would make this safe, if it can be guaranteed to return in-bounds before memory access. (llvm:7)
- The situation has not gotten friendlier to old-school pointer manipulations since https://lwn.net/Articles/278137/ was written in 2008. The pattern could still be found in code exposed to malicious interlocutors in 2013: https://access.redhat.com/security/cve/CVE-2013-5607 (regehr-blog:0)
- I suspect this happens a lot, especially with loop termination when you’re not just iterating forwards through an array.

Eg: the second version of the code in this answer: http://stackoverflow.com/a/29195143/1400793 (regehr-blog:1)
- Why do you have to ask questions that make me want to audit all of my code? (regehr-blog:7)
- I think I’ve seen ptr + ARRAY_SIZE - 1; does that count? (regehr-blog:10)
- I guess compilers assume no overflow. (regehr-blog:15)
- The same tool has a memory mapped file (i.e. mmap) where a pointer is kept most of the time in-bounds the mapped file buffer, but in some subtle cases, the pointer points one byte before the buffer. It is never dereferenced when this happens, though. (regehr-blog:18)
- I’ve never had a problem doing this in practice. As long as you don’t actually overflow (i.e. wrap around from one end of the address space to the other) then you should get away with it. I don’t think its legal in the standard, but there is so much existing code that does this kind of pointer math, that I don’t think compilers are going to mess with it.

For example, we have converted pointers within a certain address range (virtual memory region, etc.) into integer offsets and stored those with fewer bits, and then converted them back before using the pointer. As long as it points to a valid address when using it (and you are careful about strict aliasing etc.) then it seems safe enough. (regehr-blog:26)
- Isn’t this how some versions of offsetof are implemented? (regehr-blog:29)
- Think variably sized arrays. If you have a struct with an array of [1] but you’ve allocated > 1 final members .... (x11:3)
- Normal C compilers don’t spend extra runtime cycles normalizing or bounds-checking pointers. If the compiler can detect that the constructed value is illegal, it ought to say so; and I fail to see a case where assuming such a constructed value is in bounds offers any traction on anything useful.

I could imagine a feeble program checker objecting to x = p-100; x[100] = 1 if it recorded on the first statement that x is within the bounds of p, and then (if the bounds of p are less than 100) noticing that the second statement makes an out of bounds access; but such a checker ought to just object to the first statement.

There is a fair amount of string handling code that uses one-before-the-beginning pointers to avoid special cases. I rewrite this when I find it, so I don’t specifically know of any, but I’m sure lots is out there.

I would expect a program checker to reject such code. (x11:5)
- C++ (and all C code that interact with it) (x11:6)
- As I said above, I’m unfortunately gotten wind of with some rather insane behavior from compilers; and so since you ask the question, I say "I don’t know". In my normal course of programming, I would probably not think about whether a pointer accidentally became invalid in the course of some pointer arithmetic, as long as I knew it was going to be correct at the end. Compilers that assume otherwise are laying bear traps for the unwary. (xen:2)

(There was much fun to be had with XSA-55) (xen:3)
Given two structure types that have the same initial members, can you use a pointer of one type to access the initial members of a value of the other?

Will that work in normal C compilers?

- yes : 219 (69%)
- only sometimes : 54 (17%)
- no : 17 (5%)
- don’t know : 22 (6%)
- I don’t know what the question is asking : 4 (1%)
- no response : 7

Do you know of real code that relies on it?

- yes : 157 (50%)
- yes, but it shouldn’t : 54 (17%)
- no, but there might well be : 59 (19%)
- no, that would be crazy : 22 (7%)
- don’t know : 18 (5%)
- no response : 13

Comment

- Windows, and other software, does it all over the place – struct has an initial member determining size or type which defines how the rest of the struct is to be read (main2.2:0)
- the compiler is free to layout the members in any way within the memory of the struct. No guarantee that the first member is the first in memory. (main2.2:3)
- given that structure padding is taken into account or managed somehow, it’s completely legit to do this.
- GTK api does this to mimic an Object-Oriented environment in plain C (with nested structs) (main2.2:5)
- Notification messages on Windows. A WM_NOTIFY handler first works with NMHDR* and then (for some particular notifications) casts it to other struct (which begins with NMHDR*).

Maybe used in gtk+ too for "class" inheritance, but not sure about that right now. (main2.2:6)
- All sorts of OS code that has some standard linked list at the head of a struct for example. Many examples of this. Poor man’s object-oriented programming in C. (main2.2:8)
- This kind of thing is used to do ‘inheritance’ in C.

I’m not sure if this falls foul of strict aliasing or not.

I would have to check the rules carefully before I relied on this. I think probably this construct is safe:

```c
struct base
{
    int original_member;
};

struct derived
{
    base parent;
    int additional_member;
};
```

But this probably breaks the rules:

```c
struct base
{
    int original_member;
};
```
struct derived {
    int original_member;
    int additional_member;
};

(main2.2:9)
- Yes, IIRC this is explicitly allowed by the C99 standard, and I personally have written code that relies on this language feature.

(main2.2:10)
- This is poor man’s inheritance for C. If you have a family of structures that all have the same initial members, that should work and lots of stuff is going to break if it doesn’t. (main2.2:18)
- Padding could be different if that’s somehow a worthwhile optimization, I wouldn’t risk. Or maybe after checking that pointer-to-members are equal if I *have* to. (main2:7)
- If the compiler can prove that the types don’t match, it may take the opportunity to optimize away the access as "undefined". (main2:8)
- I’m not sure how much this survey cares about C++; there it’s common (main2:9)
- LLVM’s hand rolled rtti does this! (main2:10)
- Assuming you mean two unrelated types not casting a struct A* to a pointer to its first member (main2:12)
- It’s fine since there can be no padding before the first element of a strict (main2:14)
- The FreeBSD kernel and many other things do this. Most anything that uses structs to access IPv4 and IPv6 header data. (main2:16)
- The right way to do this is to have the initial member of both structures be a 'struct shared_stuff' and cast to that type. (Which is safe, because you’re allowed to cast to the type of the initial member.) (main2:18)
- This sounds like -fno-strict-aliasing. (main2:19)
- Sounds like a common way to do inheritance before C++. (main2:26)
- This is very common. It is often achieved by simply making the first member of the second structure an instance of the first structure, but in some cases (e.g., the Berkeley socket address types) even dissimilar views to the same representation data are used at different times. (main2:27)
- Lots of code uses this type punning. (main2:29)
- This happens all the time. Not just restricted to initial members, using the CONTAINING_RECORD() macro. (main2:30)
- This is an accident waiting to happen. Imagine that the code references the second element of the struct. A subsequent code change modifies the type of the first element. The offset to the second element will no longer be consistent, but that is not readily apparent. (main2:32)
- My mental model is that the layout of the structs is identical so this shouldn’t matter, even if optimizations have fired (main2:34)
- Pointer arithmetic within an array of similar structs? (main2:35)
- Disabling strict aliasing as is possible with eg gcc makes this work reliably. (main2:46)
- It violates the strict aliasing rule if it is accessed using both types of pointers. (main2:48)
- I’m fairly certain this technique occurs in the Julia implementation. (main2:49)
- I don’t think that padding bytes are specified that definitively. For that matter, I don’t know for sure if byte-wise layout is specified definitively. (main2:55)
- I recall the rules for "union" specifically allow accessing both views of the union as long as the access is the initial members that are the same on both sides. (I know that’s not the question you’re asking. Just related)

I’ve seen code that does things like the small-string optimization, where they’ll treat a pointer to the string object either as a pointer to a length/ptr pair, or, if the length is small enough, to a length followed by a char array. I’ve never understood why it’s important to break such a use case. (main2:59)
- assuming that the structures are layout-compatible. (main2:64)
- This is not guaranteed to work unless the two structures are members of a union...
ion, and the object is an instance of that union type. But the things that compilers do to make that case work will usually also make the non-union-member case work. (main2:69)
- I think this is allowed by the standard explicitly, but I could be wrong. I'd check but that would be cheating. My other answers about the standard are from memory. (main2:70)
- Shouldn't make any assumptions about how a compiler lays out your structures. Also, "initial" members? (main2:72)
- if they have the same padding (main2:73)
- Yes, this is permitted by the std. (main2:74)
- It works, maybe it's hacky but it works (main2:75)
- This violates the strict aliasing rule. (main2:80)
- I see this when people want to homebrew "object orientation" with structs, like minimal struct inheritance, using pointer casting to take a pointer to a member of a child struct and use it to refer to the member of the parent struct. (main2:81)
- A common example is using the two structs sockaddr and sockaddr_in in the socket API (main2:88)
- Many hand-made OO libraries for C rely on this. (main2:90)
- Guaranteed by the standard only if the structures are members of the same union (clause 6.5.2.3, structure and union members) but it will normally work for bare structures.

Very common for implementing object-oriented polymorphism, e.g. in bytecode interpreters. (main2:94)
- I can swear I've seen this in both Windows headers and the Linux kernel. (main2:96)
- This is a common idiom in X11 event handling code - you are forced into it by the XLib API which assumes that you can read the event type from the first member of the XEvent union regardless of which subtype of the union will be used to read the rest of the data. (main2:99)
- Allowed for union members, as far as I can see forbidden otherwise. Which is a pain. (main2:100)
- Same as above: object code in C. (main2:102)
- This is extremely common and I believe safe in the degenerate case that the castee type is the initial member of the other struct. (main2:106)
- A common 'pattern' for implementing struct inheritance in C is to include the base struct as the first member of the derived struct and casting pointers. (main2:108)
- This is used so commonly that no compiler would dare to do anything than what you expect. (main2:115)
- That's how the original C structs worked anyways. Struct fields were in the global namespace, and were basically just an offset. (main2:121)
- This is a clear violation of strict aliasing rules but I have seen it in real code. The most common alternative is to copy the data, field by field, from one type to another. This is expensive. (main2:123)
- (main2:124)
- Too much is assumed about object layout. (main2:127)
- It should be OK by the spec if one struct is embedded as the first field of the other. (main2:129)
- poor man's class system (main2:134)
- Half of the Win32 API, BSD sockets and most OOP done in C would break. (main2:137)
- Structure layout had best be guaranteed to be consistent given the same list and order of structure components, or else structural inheritance techniques will not work. (main2:138)
- Been bitten by this personally. A later member can cause the alignment/padding of an earlier member to change. (main2:139)
- I am not entirely sure whether the C standard tells anything about structure layout. In my experience, it is always in the specified order. And trouble only starts with padding of subsequent fields. (main2:144)
- Assuming the same compiler (on the same platform) with the same padding settings. It's likely to work. (main2:147)
- C polymorphism/inheritence, reinterpret casting, etc. (main2:148)
- If the one is the first field of the other. (main2:156)
- This is used by at least the standard library net address sockaddr*_t structures. (main2:157)
- It would probably be better practice to define these two different data structures using a common struct type as their first element. Then a consistent pointer type could be used. (main2:158)
- I do remember that the standard’s aliasing rules allow one safe way to do this, but I don’t remember if this is the safe way. If I wanted to write such code, I’d look up the aliasing rules and do it the right way, whatever that is. (main2:159)
- Any dynamically-typed language interpreter does this, eg. See lua language.

Also I’ve seen code that stores magic numbers as the first word. It’s accessed using a structure but instances of the structure in an array are skipped if the magic number isn’t there. (main2:163)
- Object oriented programming in C: routinely used to access members of the base class with a pointer of the derived class. (main2:173)
- Strict aliasing (main2:176)
- Isn’t this the Berkeley sockets API for struct sockaddr? (main2:177)
- struct sockaddr, struct sockaddr_storage, etc are routinely used this way even though I don’t think the standard allows it unless the structs have the nonstandard __packed__ attribute. (freebsd:0)
- So much. (freebsd:1)
- Having a common header struct is a common idiom (especially for simulating variant object types - X11 code has lots of common header structs) but you should only access struct members via the correct type. (freebsd:6)
- This exception is only guaranteed when both are members of the same union and access is through the union. I think I’ve seen compilers warn about such accesses, but I’m not sure I’ve seen them change behavior for them, yet. (freebsd:8)
- This is something that GCC tends to actually kill in practice (if strict aliasing is on); I’ve had to fix bugs that were caused by it. (gcc:3)
- A type error, surely? (gcc:4)
- This is kind of like "inheritance in C" Often this was used to read the header of a message of a network interface and then to use a case statement to select the appropriate struct for the body of the message that was received. (google:1)
- Used for mimicking inheritance in plain C. (google:2)
- doesn’t that happen with the linux kernel linked lists? (google:3)
- I would say only initial member (not plural), padding / alignment of members in each struct may be defined differently, so only the first member is defined to be aligned equally. Should not be done, too much subtleties, beware classes having (or getting) virtual members and a vtable, etc. Other than plain vanilla structs programs should not make assumptions on class layouts or padding other than what is well defined or explicitly set. (google:4)
- that’s old school C "polymorphism" and it works as long as one is careful about aliasing within a single function. No idea what global optimizers do to it, but I’d imagine it can deal with it because it would break a lot of existing code. (google:5)
- This is guaranteed for separate structures that are part of unions, but not for structures in general. (google:7)
- Python does this, as mentioned above.

I have a feeling this behavior is explicitly protected by the spec. (google:11)
- OOP like code in C uses that often. (google:12)
- sockets (sockaddr)
- implementation of vtables (google:13)
- UN*X sockaddr works like this! Also other UN*X-y ioctl's (google:14)
- This is how inheritance works in C. Granted, this is usually done by embedding one type into the other. Giving the two structures the same data members with out doing this is bad practice for code health reasons, above anything else. Also, it’ll aggravate the padding issue discussed previously.

In particular, I see this used in place of a tagged union, where size of represented data varies. I usually see the duplicated members encapsulated in a struct when this tag is a simple integer, however, I have seen this omitted. (google:16)
- Inheritance. (google:22)
  - Padding or other attributes for the structures can be different, in which case this definitely wouldn’t be possible, but it could work in other cases. (google:26)
  - Reading structures that have a size field. Check the size, free, then re-read into the right sized memory. (google:27)
  - As long as you’re going through sufficient levels of type casts, this doesn’t seem like anything should go wrong. (google:29)
  - struct sockaddr (google:30)
  - You’d need a bit of casting, but it can be used to implement parent and child structs. (google:31)
  - Structure packing of the 2 structures could be different. (google:32)
  - I don’t know if this is legal, but it seems to be portable as I have seen it in a large number of codebases and don’t remember any issues with it. If the structures were completely identical I think it would always be legal. (google:34)
- Often this is used for a pseudo object-oriented inheritance system, where child objects are required to have the full layout of the parent at the beginning. However I’m not 100% convinced that this is guaranteed to work if the structures aren’t carefully padded to lie on appropriate word boundaries. (google:35)
  - Haha, the good old C++ #define private public hack. Sometimes works. Nobody should ever rely on that.

However, I am aware that Xlib relies on this for XEvent (so you can use an event as an XAnyEvent, do some processing, and then use its actual type), and due to lack of better built-in inheritance mechanisms in C, I can’t claim a superior solution without any drawbacks. At least Xlib has the type tag outside the union. Also, Gtk object handling may be doing similar things (didn’t check). (google:37)
- Depends on the structure packing rules in play. If you can control those (usually with #pragmas), this can be safe. (google:38)
  - I believe I saw this in BrewMP’s object system. It was used to implement subclass classes. There would be a macro that defined initial members. It was used by both the base class struct and the sub-class structs. (google:40)
  - In C, the type of a pointer is what the programmer tells it to be. The compiler doesn’t check the type at runtime. This is why C++ introduced stricter casting rules. (google:41)
  - syscall implementation data structures can have padding, e.g. struct siginfo_t. (google:42)
  - This is a very common thing when you have some collection of variable records, such as struct type_a {u8 type; u8 length; u8 type_a_field[20]}; and struct type_b {u8 type; u8 length; u32 type_b_field[2]}. Usually the best way to write this is struct header {u8 type; u8 length;}; struct type_a {struct header h; u8 type_a_field[20]};, but not everyone does it that way and just overlapping different structures should work as well. Structures define memory layout exactly. (google:43)
  - yes (for what?): IIRC Python’s C API uses this trick for defining objects. (google:45)
  - Struct sockaddr_storage and sockaddr_in6 on any POSIX (google:47)
  - Very common, e.g., all the socket struct type punning. A compiler would be crazy to make it not work. (libc:1)
  - union can be used for this, but mere pointer cast is aliasing violation.
  - i’m sure this is commonly relied on, pre-c99 code has lot of aliasing violations.

it works across boundaries where the compiler cannot see (eg passing a struct pointer from userspace to the kernel), but within a library written in c such assumption is not portable and i’d expect compiler optimizations to break it without special annotations. (libc:3)
- Used all over the place for standard network code. You cannot break this without completely breaking the C socket library. This is C’s method of doing polymorphism. (libc:4)
- Different structure padding. (libc:5)
- Dang it, just use nested structures, or a union. Working within the language spec here isn’t hard.

That said, the existence of such code is why production compilers have to carry around -fno-strict-aliasing and their moral equivalents. (libc:6)
- necessary for inheritance in OO-style C code, declared legal in latest C99 amendment anyway. (llvm:4)
- This is used all over the place. (llvm:5)
- Wow, C is a really broken language. Or it’s intentionally malicious; giving developers enough rope to hang themselves. One of the two. (llvm:7)
- According to the standard, structs could have all the same members and still be incompatible because they do not have the same tags. In practice, ABI constraints (that are not considered at all in the standard) sometimes make this sort of code work. (regehr-blog:0)
- Does cpython rely on this? I looked a few years ago and my recollection was that their object headers depended on this.

Of course, it might have been cleaned up now, or my recollection might be faulty. (regehr-blog:1)
- I prefer using an enumerated union for generic typing. (regehr-blog:7)
- I think the Ruby interpreter uses this extensively to implement their objects. Not totally sure.

https://github.com/ruby/ruby/blob/trunk/include/ruby/ruby.h (regehr-blog:8)
- This is effectively duck typing. (regehr-blog:9)
- used all the time in networking for packet header layout. (regehr-blog:11)
- One possible problem with this scheme is triggering an alignment fault exception. (regehr-blog:14)
- Yes, I see this all the time with signal structures.

Sometimes there is a union containing the structures, but of the pointers are just casted. (regehr-blog:15)
- I think this case is common when a OO-style programming is implemented in C. (regehr-blog:18)
- I believe this is done in the Linux kernel. (regehr-blog:20)
- all the object-in-pain-c libraries (regehr-blog:21)
- This is common with data structures that start with a small header that describes how the rest of the data is to be interpreted. Its not standards-compliant but as long as you lay out the struct members in a predictable way it works fine on every compiler I am aware of. (regehr-blog:26)
- seen this used to implement a 'poor man's' polymorphism (regehr-blog:29)
- Lots of X11 and OS code, trying to emulate object class/sub-class models. (x11:0)
- Used for the struct-in-struct inheritance pattern (x11:2)
- In the kernel you often cast a pointer to your softc when the first memory of that softc has a more generic softc as it's first member and the pointer to that generic softc is used as an argument in callbacks. (x11:3)
- Should work if the same padding / packing / alignment / struct reordering rules apply (x11:4)
- This is explicitly blessed by the C standard. (Perhaps regretfully, but it is so.)

The most well-known example is the family of socket address structures in Unix: struct sockaddr, struct sockaddr_storage, struct sockaddr_in, struct sockaddr_in6, struct sockaddr_un, etc. These are not going to go away, so whatever you're doing needs to deal with them... sorry. (x11:5)
- sockaddr/sockaddr_in/sockaddr_in6 (x11:6)
- DLPI message structs all have a common dl_primitive field as their first member. Code handling them relies on being able to decode the first few bytes of an M_PROTO message as this dl_primitive field such that the whole message can then be cast to the correct struct for further dereferencing. (xen:0)
- There are many reasons for which C programmers assume that they can rely on the layout of data structures in memory. I can certainly imagine someone taking advantage of that fact to do something like a Java "Interface": i.e., whatever my struct is, some bits of it look like this so I can access them no matter what.
The "proper" way to do this of course would be with some sort of union or sub-element, but I can imagine someone doing this. (xen:2)
- I would expect that it will most likely work in the general case, but absolutely can’t be relied upon. (xen:3)

Can an unsigned character array be used (in the same way as a malloc’d region) to hold values of other types?

Will that work in normal C compilers?
- yes : 243 (76%)
- only sometimes : 49 (15%)
- no : 7 (2%)
- don’t know : 15 (4%)
- I don’t know what the question is asking : 2 (0%)
- no response : 7

Do you know of real code that relies on it?
- yes : 201 (65%)
- yes, but it shouldn’t : 30 (9%)
- no, but there might well be : 55 (17%)
- no, that would be crazy : 6 (1%)
- don’t know : 16 (5%)
- no response : 15

Comment
- Marshalling (main2.2:0)
- Array is just a region of memory. Can hold anything, but you need to fiddle it in and out.

Example: EEPROM handler copies memory (char array) from and to NVRAM (main2.2:3)
- it’s useful for serialization (main2.2:5)
- for so many things. E.g.
  -- for generic functions similar to memcpy()
  -- for serialization and deserialization
  -- for computation of hashes of the pointed structure (main2.2:6)
  - All kinds of custom allocators. Or passing some buffer of some number of bytes to an API, that puts whatever in there. The API only tells you how many bytes the buffer needs to be. (main2.2:8)
  - Possibly only strictly legal if you use plain ‘char’, but yes I would expect this to work.

I have seen this used in C++ templates to manage lifetimes and avoid memory allocations for a single member object of arbitrary size. (main2.2:9)
- While aliasing and observing the representation of an arbitrary object through a pointer to (signed, unsigned or sign-unqualified) char is explicitly defined, the opposite is not true. It violates the strict aliasing rule. I, however, know of code that uses vendor-specific alignment qualifications to char arrays and relies on the lack of strict aliasing-based optimizations to do tricky things. (main2.2:10)
- this might give trouble with type based alias analysis? (main2:0)
- unsigned char[] is the one type that can hold anything except trap values.

unsigned char[] ought to be safe, or nothing’s sacred. (main2:7)
- Many serialization / communication buffers work like this. This also works with char arrays, not only unsigned char. (main2:8)
- Providing alignment is made OK, I do this. (main2:9)
- This sounds extremely reasonable, but I can’t think of an example of this. (main2:11)
- BSD kernels use the caddr_t typedef for allocations that will be manipulated as bytes. (main2:16)
- Not sure what you mean by "values". If you mean memcpy or otherwise
operally aligned way) putting data into a uchar [], then plenty of code does this like anything that serializes to a network buffer. (main2:20)
- yes, have seen custom memory allocators that did this - have to take care wi
th alignment, though (main2:28)
- Lots of code either allocates via malloc a chunk of memory, or uses a small
uchar8_t array on the stack to create ‘objects’ that are then sent to other thin
gs (like say arp packets) (main2:29)
- Encoder/Decoders do this all the time. They read bytes from a file into an
unsigned char buffer, then cast a struct * on top of it to pick out the relevant
fields and move on. (main2:30)
- The architecture may have alignment constraints enforced by malloc() but not
by static allocation. e.g. The malloc() function might always return a value th
at is a multiple of the architecture’s word size. (main2:32)
- Care is required about alignment. (main2:37)
- It will work on platforms without strict alignment. (main2:38)
- There’s technically nothing wrong with this. As long as you are doing it int
entionally. (main2:40)
- char pointers are allowed to alias any type. (main2:48)
- Must be marked to require appropriate alignment. (main2:52)
- Yes, for the "small string optimization", such a thing would be useful. (mai
n2:59)
- is only char[] permitted for this? (main2:61)
- assuming that the storage is aligned and the type is a "POD" type. (main2:64)
- no alignment guarantee (main2:68)
- I suspect this is okay as long as you have control over the alignment, becau
se pointing other type pointers into the array would need to be correctly aligne
d for that type. (main2:70)
- This is an _extremely_ common way of doing stack buffers. (main2:72)
- Don’t forget to align it. (main2:74)
- a char is 1 byte wide, so it has completely sense (main2:75)
- The other way around is OK. (main2:80)
- This might be architecture specific, for some alignment on datatype size ins
ide of the array is required. (main2:87)
- char’, ‘unsigned char’ and ‘signed char’ being three distinct types (as oppo
sed to having just two types if we substitute short for char) is so obscure that
I don’t even know how to explain the rationale to a novice except by something
equivalent to "that’s how it is, deal with it". (main2:90)
- No, not to hold values, but they can be used to copy the representation of v
alues of other types. Thus, you can copy the bytes from an unsigned char array i
nto a float * variable and use the float * variable (as long as the original byt
es came from copying a float * variable into the unsigned char array). (main2:92)
- You probably need to be careful about alignment, though (either via an attri
bute to make the array 4 or 8 aligned as needed or by manually finding the first
4/8 aligned address). (main2:93)
- Guaranteed by the representation of types clause. (main2:94)
- If alignment constraints are upheld, or the char array is not used as that s
truct but only for memcpy(), it’s safe. (main2:96)
- Caveat that you need to get alignment right somehow. (main2:100)
- Special memory allocators come to mind immediately. (main2:102)
- The array needs to be properly aligned for the target architecture and type.
(main2:103)
- With the caveat of appropriate alignment, this is how malloc is often implem
ented in embedded systems. (main2:106)
- Alignment issues might arise at run-time, but it would compile. (main2:107)
- You need to make sure of correct alignment for the contained type, though.
(main2:108)
- You often need to be careful about alignment. Natural alignment is often nec
essary, especially around SIMD extension types and intrinsic functions. (main2:
111)
- This depends on the alignment restrictions of the object. (main2:115)
- I would expect it to work with most compilers. I haven’t seen it done and I’
m not confident that it would work. (main2:123)
- It’s simply unavoidable sometimes, especially with crypto-related code. (main2:129)
  - embedded software: unsigned char array are used as byte storage for unknown
    content, or content that will be decommuted by other components, assuming that t
    he char is 8bit (main2:133)
  - "char" predates "void"; it is often still used. (main2:134)
  - Weird cases of architecture dependent issues arise when counting on the size
    of unsigned char to be equal to 1 byte. It also seems like a bad idea with sta
    ck vs heap memory usage though I honestly don’t know the details. (main2:141)
  - implemented simple malloc blocks this way (main2:143)
  - It might be unwise to think you can write a better memory allocator than the
    library, but if the library doesn’t do what you need go ahead! (main2:147)
  - scratch space, heap allocators, ring buffers (main2:148)
  - I’m kinda losing patience with these questions. It can be cast to do so and
    millions of programs do. Absent casts, maybe, maybe not, I don’t think anyone fa
    cing this problem pauses for a second to consider whether the cast is breaking a
    spec rule or just quieting an overzealous compiler. (main2:156)
  - This is historical behavior from the first libraries that returned char* fro
    m malloc routines, etc. (main2:157)
  - It is possible to serialize other data into and out of arrays of unsigned chc
    ar. You must be careful of alignment and careful of portability concerns. In som
    e microprocessors you can violate alignment requirements for different types of
    memory accesses and cause a crash; for example, on the 68000 if you access a 16−
    bit word at an odd address, you will trigger an address exception. So it would n
    ot be safe to take an arbitrary address of a byte in an array of unsigned char a
    nd write to it by casting it to a pointer to, say, a short, int, or long. Howeve
    r it is generally safe to work the other way, using a cast from the address of a
    wider type to an unsigned char * and stepping through memory byte-wise. Keep in
    mind that any such code is likely not portable although it can be made somewhat
    more portable by using sizeof(). (main2:158)
  - Isn’t it char arrays that are allowed to alias with anything? I’d guess tha
    t unsigned char arrays don’t have that special case. (main2:160)
  - We don’t use malloc so that is the only way to do custom allocators. And als
    o, surely malloc (assuming no intrinsic) is doing that? (main2:163)
  - Data alignment should be preserved, e.g. malloc guarantees alignment for lar
    gest primitive data type while stack−resident arrays are able to not obey. (main
    2:164)
  - You just have to use casts appropriately to assure (or evade) the compiler’s
    type checker. (main2:165)
  - A void* pointer should be used instead, and values extracted through casting
    , but everyone uses char* instead. (main2:173)
  - Not 100% sure about _unsigned_ char. (main2:176)
  - I think it requires −fnostrict−aliasing. I myself have done this when I nee
    d to maintain compatibility with a legacy API. For example, a function that tak
    es a uint8_t* argument but then casts it to uint32_t*. (main2:177)

Also, the formerly open source Likewise CIFS server contains copious casts betwe
en uint32_t and uint8_t[4]. (freebsd:0)
  - Assuming appropriate alignment restrictions are maintained and the array poi
    nters are appropriately cast, this should work. (freebsd:6)
  - The obvious problem cases are ones where the target type requires stricter a
    lignment. Assuming you’re not asking about cases where a value is overlaid by a
    character array, as with a union. (freebsd:7)
  - The rules about the effective type of an object are a bit hard to follow. (f
    reebsd:8)
  - If nothing else, alignment must be ensured, and there is no way to do this p
    ortably except possibly with some C11 _Alignas constructs. The non−portable meth
    od of casting to uintptr_t and using the low bits to compute the offset needed f
    or alignment works in practice though. (gcc:2)
  - Some types might have alignment requirements not satisfied by the char array
    . (google:0)
  - Depending on the platform’s memory alignment model an array of characters ma
    y or may not be properly aligned. (google:1)
  - Used for the heap of a virtual machine. (google:2)
  - No idea what you asking? Any memory area occupied by any var can be used for
storing any arbitrary data / type, alignment is up to the user to make sure it
does not mess up, but why is this a question? (google:4)
  - old-school malloc used to return character arrays...
  (google:5)
  - Some people have implemented "arena" code that works this way. Our "char" i
    s unsigned by default... is this question meant to specifically refer to the "un
    signed char" data type? (google:7)
  - This is basically the simplest form of serialization. As long as the data is
    n’t intended to be portable across platforms/compilers, and as long as the data
doesn’t contain pointers, this works fine. It’s historically very common, but gr
    adually becoming less so in the interests of portability. (google:11)
- Yes, but it must be aligned. Used e.g. to write custom allocators; I think t
  here’s even one such in K&R. (google:14)
- I’ve seen a LOT of code use this for buffering raw data before writing it to
  files. There’s just no reason for this not to work, although methods of moving
  between these formats are sometimes questionable (unaligned data types and mistr
  eatment of floating-point memory prevail). (google:21)
  - Useful when you need to do some byte-level pointer arithmetic. (google:22)
  - De / Serializing a struct. (google:23)
  - Poor practice. Intent is unclear. Size of struct may change you you would
    learn about it in production. (google:27)
- Yes but you should manually take care of alignment (google:30)
- Fixed length keys often will do this (google:31)
- custom-built memory managers (google:32)
- for what: reading packets from the network into a buffer, and interpreting t
  hat buffer as a structure. Also for making a custom allocator.

If the array has the correct properties (alignment, writability, size including
padding) then this works well, though I don’t think the standard enumerates the
possible things you need to take into account (maybe some systems allocate certa
in types in different areas of memory). To get around that I have used a union
of the different possible types with a char array big enough for any of the type
s. (google:34)
- A generic circular buffer style system, where arbitrary structures are inser
ted into the buffer. Alignment issues can make this slightly tricky though. (goo
gle:35)
- IIRC, C99 decrees that all objects can be represented as an array of bytes.
  (google:36)
- Can’t pinpoint any specific instance, but I’ve seen people allocating one ar
ray of "appropriate" size and then storing one of multiple structs in there.

They should just use an union instead...

but usually this should work, as long as the size is actually correct (the most
common pitfall) and alignment is fulfilled (this is where this would actually br
eak in practice – e.g. aligned SSE instructions may crash when accessing such da
ta). (google:37)
- Unsigned character arrays are guaranteed to be contiguous, so you can use th
  em as arbitrary buffers. (google:38)
- One of course need worry about alignment concerns, but memcpy() etc could be
  used to read and write arbitrary types into such a region. (google:39)
- I wouldn’t expect unsigned char* to behave much differently than char* in mo
  st compilers. (google:40)
- One unsigned char is usually a shorthand for one byte. If you need to store
  an image, for instance, or another byte stream, the data is typically typed as u
  nsigned char. (google:41)
- You need to be able to keep the compiler from assuming there is no pointer a
liasing.

This is used for network protocol implementation. Maybe people should us a union
instead. (google:42)
- This is the most common way to declare a stack- or BSS-allocated buffer for
  anything. (google:43)
- only sometimes (when?): It will work if you take care to align it properly.
  (google:45)
- i think it works with current compilers if the alignment is taken care of.
You sometimes see this when encrypting chunks of memory, since encryption operations work better from a C type-checking perspective when done on unsigned characters. (libc:4)

Arbitrary byte storage. (libc:5)

Presuming here that the code doing this also cares for alignment issues. (libc:6)

I’ve seen code which receives fixed size messages (from disk or network) into such an array, and then later casts the array to a struct to unpack the data. (libc:7)

This is the loophole provided for type punning. (llvm:2)

We do it all the time (llvm:3)

If done correctly, it is using sizeof (and/or offsetof). (llvm:5)

This is just asking to be unportable. (llvm:7)

Yes it can hold them (eg: by memcpy’ing their bytes), but certainly not always use the value (eg: by casting a pointer) because of alignment. I’d guess it was as always UB, but I’m not sure. (regehr-blog:1)

This is explicitly allowed by the standard, isn’t it? (regehr-blog:9)

Seen this with unions to cast something to an array of bytes (pretty sure that’s illegal too...) or to store a struct on the stack before writing that struct to flash. (regehr-blog:10)

seems like the sort of thing that would be done in OS bootstrap paths before malloc or equivalent is available. (regehr-blog:11)

I think this is OK as long as alignment requirements are met, but I’m not sure. (regehr-blog:12)

The array wouldn’t have to start on any particular alignment so your constructed type might have to start in the middle of the array to be properly aligned. (regehr-blog:14)

I often see this used to implement custom allocators. (regehr-blog:15)

Well, it should be a non-signed (not signed or unsigned) character array I think. (regehr-blog:18)

I have seen this technique used for inspecting the representation of types, for example. It’s quite common for code to take advantage of the fact that there is no trap representation for unsigned char. (regehr-blog:20)

-c is lower level language. Either char, unsigned or signed, is just a synonym to one byte. (regehr-blog:21)

You have to align the storage properly for the type you want to put in it, and you have to be careful about the strict-aliasing rule, but this works fine on all compilers I’ve tried. One example would be a dynamically-resizing array container which can store a small number N of items inside itself, switching to malloc’d storage when N is exceeded. Handy for allocating container storage on the stack (use alloca and cast it to the struct with the char array in it, and may be adjust to align for your type, and then cast the address of the char array to the type you want to use it as.) (regehr-blog:26)

We use any memory for any types, not limited to unsigned chars. E.g., an array of pointer-sized integers to hold integers and various kinds of pointers. (regehr-blog:28)

the ‘beauty’ of c casts. (regehr-blog:29)

I think this is the bit in the C spec about the compiler can assume char* aliases anything... (x11:2)

Mostly code that is derived from vax/intel that ignores unaligned accesses. Otherwise you use arrays of longs to do the same for maximal alignment. (x11:3)

accessing video memory in 8bit colour for example (x11:4)

That this is formally prohibited by the strict aliasing rules is a bug in the C standard. IMO, but I’ll defend that opinion if it comes to it.

Therefore, it works as long as the compiler doesn’t get to see it too closely. If it does, it can break arbitrarily. Allocating the space in .bss in an assembler file is a fallback position.

One does have to be careful about alignment, but that can be handled in various ways.

There are various cases where one might want to use an explicit array in place of a malloc call: chiefly either where malloc is slow, where malloc might fail, w
here malloc is unsafe due to locking/interrupts/whatnot, where malloc isn’t available yet during initialization (e.g. early in kernel boot, in a dynamic linker), or where malloc flatly doesn’t exist (embedded systems, some kernels)...

I would strongly encourage you to explicitly support this in whatever you’re doing. (x11:5)
- most networking apps. struct ip. (x11:6)
- This is common in network drivers. Packet headers are read into char arrays and then cast and dereferenced as structs (with suitable endianness swaps where necessary). (xen:0)
- I’ve seen this being employed to statically allocate memory on systems that do not have malloc() or where you must not use malloc() due to unpredictable timing (i.e., real-time systems). (xen:1)
- It’s not at all uncommon to read data in from disk and then have to interpret it; the most sensible thing to do frequently is to cast the bit you’re looking at into the pointer of the type you know it is. (In fact, I’m not really clear how else you would do this.) (xen:2)
- Strictly speaking, I believe only char arrays have this property in C, and it is implementation defined as to whether char is signed or unsigned. (xen: 3)

Despite this, it is very common to have uint8_t arrays for arbitrary data. (xen: 3)

---

Null pointers from non-constant expressions
Can you make a null pointer by casting from an expression that isn’t a constant but that evaluates to 0?

Will that work in normal C compilers?
  yes : 178 (56%)
  only sometimes : 38 (12%)
  no : 22 (6%)
  don’t know : 67 (21%)
  I don’t know what the question is asking : 11 (3%)
  no response : 7

Do you know of real code that relies on it?
  yes : 56 (18%)
  yes, but it shouldn’t : 21 (6%)
  no, but there might well be : 113 (37%)
  no, that would be crazy : 63 (20%)
  don’t know : 50 (16%)
  no response : 20

Comment
  - NULL might or might not be 0 (main2.2:3)
  - compiler should issue an implicit conversion error from int to void* (main2.2:5)
  - Sounds like UB (main2.2:7)
  - Converting opaque references back to pointers. (main2.2:8)
  - If you want a null write null.

However yes I would expect this to work, even though it’s not very useful:

void* p = malloc( 10 );
assert( (char*)p - (intptr_t)p ); (main2.2:9)
- No, because a NULL pointer’s representation need not be 0 at runtime. Almost all C code I have read does assume that e.g. (int)NULL is zero and (void *)(non ConstIntThatIsZero) results in a NULL pointer. (main2.2:10)
- again, I don’t know what the standard says about this; I can very well imagine an environment where the NULL pointer does not consist of all zero bits, and where this kind of thing breaks (main2:0)
- Too many things would break if this didn’t work. (main2:7)
- Never done this (main2:9)
- Not all nulls are the same. Null point to member function on msvc!
Also, which standard? Pre or post C++11? (main2:10)
  - Though I don’t see what use it would be (main2:26)
  - NULL was until maybe C99 or so only *conventionally* zero, and on some embedded platforms it in practice had a nonzero value. I have not seen this in a very long time. The most common offender is the ‘if (pointer)’ check for NULL. (main2:27)
  - void *p = 0;
  uintptr_t i = (uintptr_t)p;
  void *q = (void *)i;
  p == q == a null pointer.

There’s lots of code that depends on this. (main2:29)
  - The compiler is entitled to use a different value for a null pointer and coerce the constant zero to that value. e.g. It might choose a value that cannot reference a real memory location. (main2:32)
  - Compilers should not allow that without (void *)(intptr_t) cast, but most likely some compilers or some compilation flags do allow that. (main2:38)
  - why would anyone make a null pointer except with NULL? (main2:48)
  - KAI C++ compiler generated C code that relied on this/ (main2:49)
  - I don’t know of any code that does this, and can’t imagine why it would serve any purpose to write such code. (main2:59)
  - If the implementation’s null pointer representation isn’t zero, you won’t get a null pointer. (Some embedded compilers use a non-zero null pointer so they can point it at unaddressable memory, when the zero page is addressable.) (main2:69)
  - Not sure if this is allowed or not. Sticking with "don’t know". I’ll look it up later. (main2:70)
  - It will work only if 0x0 == (int) NULL on that platform (main2:75)
  - If the question is asking whether there is *some* expression that is non-constant, evaluates to 0, and leads to a null pointer, then the answer is yes, with "(void*)0" -- but if the question is whether *any* such expression can be used to make a null pointer, I am tempted to say "no" but hesitant to commit to it since it could be implementation specific. (main2:81)
  - Tagged pointers. For example, check if a tagged pointer is null:

```c
#define GET_UNTAGGED_PTR(PTR) ((void*)((uintptr_t)(PTR)) & ~0x7))
```

if (GET_UNTAGGED_PTR(Ptr) == NULL) ...

^ coded in Google Docs, there might be bugs there. (main2:90)
  - ...I thought this was legit but now you have me worried... (main2:93)
  - This will probably only fail to work on a Deathstation 9000. (main2:94)
  - Casting things to pointers is generally a bad idea, unless you know it was originally a pointer. (main2:96)
  - While I’d hope that compilers for platforms where null pointers are all-bits-0 would do the obvious sensible thing, I don’t have much faith in them. (main2:100)
  - This isn’t guaranteed by ISO C, but the entire world relies on it in practice. (main2:106)
  - Memsetting a struct will generally get you a null pointer. However, a null pointer doesn’t have to have a zero bit pattern. (main2:115)
    - If NULL != 0, I don’t think that’ll work. (main2:121)
    - It seems common. Certain kinds of checks (like NULL function pointers) seem like they would rely on this. (main2:123)
    - I suspect the answer is no, although I’m not sure how the pointer would differ from NULL. (main2:126)
  - sounds stupid or hacky (main2:133)
  - Is this another way of asking if pointers to the same object can be expected to compare equal? It seems just a coincidence that NULL has this value or that value. On many systems address 0 is valid -- it would be an advantage to use a value for NULL which is not a valid address. (main2:147)
  - bit masking to select pointers, xor swapping (main2:148)
  - The null pointer value by definition is indicated in source code using a constant zero, although the runtime representation may not be a value of all zero b
its on a given platform. If you used an expression that is not a constant zero, I would not assume that the compiler would be able to recognize that the desired result was to generate the platform-specific null pointer value for comparison, as opposed to some arbitrary pointer difference value. (main2:158)
- I know that the standard allows the representation of a null pointer to be something other than 0, but in all the "normal C compiler"s I know if, it is 0. So since (I think) you are allowed to cast between ints and pointers, I think th at this will always work in a normal C compiler. (main2:160)
- I've seen code that stores everything as offsets from a structure that relies on arithmetic producing null pointers to terminate, etc. (main2:163)
- Cast from uintptr_t

NULL is a pointer like any other (main2:176)
- Otherwise intptr_t and uintptr_t are useless or one-way only, right? (freebsd:1)
- I'm reasonably confident that the standard specifies that the null pointer i s the constant 0, rather than an expression that evaluates to 0. Most (all?) cu rrent implementations represent NULL as a pointer with all bits set to 0 but thi s definitely isn't required. (freebsd:6)
- Seems dangerous, never cosidered it (gcc:4)
- AFAIK, NULL and 0 are of different types. (google:3)
- any expression evaluating to 0 turns into a NULL pointer when cast to a poin ter. (google:5)
- Like above, XOR linked lists and XOR swap can do this. (google:11)
- In all honesty, code should probably not be doing this, as the result is goin g to be either a null pointer or a dangling pointer. (google:21)
- Depends on pedantic switches. Is yucky thing to do. (google:27)
- Null pointer need not be all 0 bits, even though the compiler will ensure th at comparison with the constant expression 0 works correctly. (google:32)
- Null pointers have type (void *)0, but that's only the constant, not the run time value. The compiler isn't required to convert all runtime zeros to the nul l pointer value (though they could as an extension). This isn't portable but in practice most systems use the actual value 0 for null pointers so you can get a way with it. (google:34)
- Null pointers don't necessarily have the same representation as a 0; the com piler knows that a cast from constant 0 to a pointer generates a null pointer, b ut it can't do the same trick for expressions.

Most real-world platforms these days use 0 as a null pointer representation, how ever, so it'll usually work. (google:36)
- Will usually work if NULL is actually the zero bit pattern (true on most cur rent platforms).

I can't imagine why anyone would be doing that, though, but the other way round seems more plausible (comparing an uintptr_t to 0 before casting it to a pointer type). And these uses can also be trivially fixed to compare against the real N ULL. (google:37)
- I don't think NULL is guaranteed to be 0, but I don't know of any system whe re it isn't. (google:40)
- You can make a null pointer by casting anything that evaluates to zero to a void *. AFAIK, there's nothing in C that protects you from that. (google:41)
- You better be sure that NULL is zero before trying this! (google:42)
- Yes. Also, NULL == (void *)(uintptr_t)0 is always true. I don't care what th e standard says, people have relied on this for years. I need to be able to seri alize any kind of pointer and pass it to a different program unit, including N ULL. (google:43)
- I think null pointer representation being 0 is often relied on (memset initial ization of structs) and pointer representation converted to integers without c hanging the bit pattern (eg aligning pointers rely on this or passing pointers t o syscalls as long).

so (void*)(int)0 works as a null pointer in practice. (libc:3)
- This is the standard "NULL may not be all-bits-zero" thing. In practice, it basically has to be; too much stuff allocates structs with calloc or calls mems et on them, and the all-bits-zero bit pattern has to be equivalent to a NULL poi
− If casting from intptr_t to void* can produce a null, then anything will. (libc:4)
− If your implementation uses something other than zero to represent null, this will fail. (llvm:2)
− You can put anything you want into a ptr. (llvm:3)
− I can think of silly ways to do that, but none that anybody would actually do.
(llvm:5)
− /* Dumb idea */
void *p = (void*)(true == false);
printf("%p\n", p); (llvm:7)
− I assume an expression containing a call to a function that returns NULL doesn’t count as a non-const expression. (regoehr-blog:7)
− uintptr_t conversion (regoehr-blog:16)
− On every C compiler I’ve ever used, null pointers to most types are represented by an all-zero-bits pattern. So any integer expression producing zero, can be cast to a pointer type and the result will function as a null pointer. This is safe as long as the optimizer can’t statically determine the value of the expression. (Reading a global variable which is defined in a different translation unit seems to work, even with link-time optimizations). (regoehr-blog:26)
− Works on common architectures, where NULL is represented as 0, but not on some niche architectures with other representations. (x11:0)
− The code wasn’t committed (probably fortunately), but [1] attempted to fix a bug in a list-walking macro by doing "&_next->__field != NULL" where &...->__field is essentially a pointer addition.

− There is no real platform where null is not all-bits-0, and there is no real platform where an integer zero value is not all-bits-0 too. There is no interesting non-linting platform where converting an integer of the right size to a pointer changes the representation. Therefore, it will work unless the compiler explicitly breaks it, and only a linting compiler will bother to explicitly break it.

Well. On second thought I suppose a compiler could remove a subsequent test for null and thereby break it; so you’d need to write the subsequent test using a similar non-constant expression. Blah.

That said, as opposed to item 13 below, there’s no valid reason to do this. At least, I assume you’re talking about either things like (void *) (complicated_expression) that some gcc versions for a while used to allow as constants if the optimizer could figure out how to fold it, like (void *) (&a - &b) where a and b are either the same or not depending on dynamic linking gymnastics, or like (void *) ((unsigned)p * valid_flag) where valid_flag is either 0 or 1. All of these things are better done other ways. (Though dynamic linker gymnastics are ugly enough in general that there may not be a compatible or readily deployable alternative for platforms that are already doing crap like this.) (x11:5)
− I would look at the linux kernel ERR_PTR macro usage if i was to look for code depending on this (x11:6)
− I can’t off the top of my head think of a situation in which this would be a sensible thing to do; particularly as one would expect that an expression which might end up pointing to (void *)0 might also end up pointing to (void *)1, which is very rarely something you actually want.

So I’m about halfway between "that would be crazy" and "but here might well be". (xen:2)
− In DOS, the NULL pointer was checked by derefencing and looking for the value 0. (xen:3)

Null pointer representations
Can null pointers be assumed to be represented with 0?
Will that work in normal C compilers?
Do you know of real code that relies on it?

<table>
<thead>
<tr>
<th>Choice</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>187 (60%)</td>
</tr>
<tr>
<td>yes, but it shouldn’t</td>
<td>61 (19%)</td>
</tr>
<tr>
<td>no, but there might well be</td>
<td>42 (13%)</td>
</tr>
<tr>
<td>no, that would be crazy</td>
<td>7 (2%)</td>
</tr>
<tr>
<td>don’t know</td>
<td>12 (3%)</td>
</tr>
<tr>
<td>no response</td>
<td>14</td>
</tr>
</tbody>
</table>

I don’t know what the question is asking:

<table>
<thead>
<tr>
<th>Choice</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>201 (63%)</td>
</tr>
<tr>
<td>only sometimes</td>
<td>50 (15%)</td>
</tr>
<tr>
<td>no</td>
<td>54 (17%)</td>
</tr>
<tr>
<td>don’t know</td>
<td>7 (2%)</td>
</tr>
<tr>
<td>don’t know what the question is asking</td>
<td>4 (1%)</td>
</tr>
<tr>
<td>no response</td>
<td>7</td>
</tr>
</tbody>
</table>

Comment

- Not sure whether the question is asking whether NULL can be replaced by 0 in the source code (it can)

Or whether the bits of the pointer all end up as 0 (they do not have to)

- NULL might or might not be 0
- #define NULL 0
- Frequently used for initialization of dynamically allocated structs or arrays, even if those embed pointers, with memset(ptr, 0, size) or ZeroMemory() on Windows.
- A lot of code C code does implicit if(foo) instead of if(foo == NULL) comparisons
- Lots of casting from 0 integers to pointers. And mixing code with assembler, assembler assumes zero.
- This is another ‘nonstandard’ thing that everyone ignores. People assume null pointers are 0.
- I believe it’s the compiler’s responsibility to convert literal 0 to any platform-specific null pointer. Though it would be crazy to make a platform where null pointer isn’t 0 these days.
- I have a vague recollection that there once existed architectures where this wasn’t true. Certainly there is lots of code out there that relies on NULL==0 now.
- I’d say "only sometimes", but I haven’t actually seen any sane target platform when null pointers aren’t 0.

I’ve seen code where there’s valid data at address 0 that may need to be fetched somehow, though.

- I know architecture-dependent code that relies on it. If there was an architecture where null pointers were represented differently, that code would adapt, but this works fine on current mainstream architectures.
- Again 64 to 48 bit compression
- Which standard?
- (void *)0 is always NULL, but most programmers assume any value of "0" is NULL.

Lots of code relies on (void *)0 being a valid function pointer.

- I assume that by "normal C compilers" we mean "on 99.999% of machines currently in use".
- Again, some systems have unusual memory layouts.
- See above. I have seen code bases that take advantage of the fact that NULL is typically protected in practice on UNIX-derived systems with an unmapped first page of memory to define several bogus flag values "near" NULL. This is also unsafe.
- lots of code assumes that either a static allocation or a calloc() of memory results in NULL pointers.
- if (ptr) { use it }
But some segmented memory systems (IBM AS/400 IIRC) the NULL pointers isn’t actually all-zeros since the pointer bits include a non-zero segment selector, so that break much code as above.

I don’t know of any current systems where that’s actually the case however. (main2:30)
- See 12 above. (main2:32)
- Linux kernel relies on this heavily (main2:34)
- I’d have to look this one up. (main2:35)
- [Again the historical restriction here is because on 8086 some memory models had many null pointers, one per segment] (main2:37)
- In theory NULL doesn’t have to be NULL, but there is so much code that instead of ‘if (ptr != NULL)’ do ‘if (ptr)’ that it would be impossible in practice. (main2:38)
- I hesitate to write "yes, but it shouldn’t" as the code in question is only ever going to run on a system where NULL truly is 0. (main2:44)
- It compares to zero and I don’t know any code that assumes anything beyond comparable to zero. (main2:48)
- The question needs clarification of whether "represented with 0" means at the source level or binary level. I’m assuming you mean binary level. Clearly at the source level 0 is a valid way to write a null pointer, even if some antique compiler uses ~0 as its internal representation. (main2:49)
- Though in this case it’s merely status quo. I’ve actually run into cases where pointers are represented in an alternative way (e.g. a DMA controller which makes linked lists in its descriptor memory by truncating pointers to 16 bits, and representing NULL as 0xFFFF), but while I’d love to be able to explicitly specify some particular representation to be used under specific circumstances (with the compiler automatically converting between representations when needed), C’s imply isn’t that expressive. (main2:50)
- The standard doesn’t guarantee it, and I vaguely think there once was a system where null pointers were not all bits zero, but I don’t think it exists anymore. So, it’s the kind of thing that I deeply believe is wrong to rely on, but that I do no longer have a handy counter example to cite. (main2:53)
- My understanding is that (1) memset-ing a pointer to NULL is NOT guaranteed by the spec to produce a null pointer, but that (2) it does on all systems that most people care about, and that there is real code that relies on that. Being able to memset a struct to zero and have all the fields come out null/zero is convenient enough that I kind of wish the spec would change in this regard. (main2:54)
- The most common way this comes up is that people assume static data will be zero-initialized, and that in such zero-initialized data, all the pointers are Null. (main2:59)
- While not standard, I’ve not encountered any compilers using values other than zero (main2:62)
- In games, memseting the structures with pointers. In games, deserialization code for specific platforms doesn’t patch NULL pointers because we know they are 0x0000_0000. (main2:64)
- Depends on how your if (!ptr) is implemented, I suppose... (main2:66)
- People love to memset structs to 0. Same comment as previous question: some embedded systems don’t use 0 because it’s a valid address. It’s not unreasonable to rely on this if you don’t expect your code to run on such an oddball system. (main2:69)
- All modern systems that I know of use all bits zero null pointers even though the standard permits other representations, so I think it’s a valid assumption. I would avoid it personally as I don’t think it buys much. The exception might be if I can prove a large performance gain. (main2:70)
- The std doesn’t guarantee this, but non-zero null is crazy. (main2:74)
- (void*) 0 is ensured to be equal to NULL, but still i’ve doubts about the need of having (int) NULL == 0x0 (main2:75)
- nullptr seems a better solution to me. It has a distinct type that permits to catch more errors at runtime. (main2:78)
- Probably depends on the platform. (main2:79)
- The standard does not restrict what null pointer constant must be. There might be some strange platform where NULL is not defined to be 0, but these days it is very rare to see such a platform. (main2:80)
- It will compare equal to 0, but does not have to be represented by all zero
bits, and behavior upon casting to integer is allowed to be implementation specific. (main2:81)
- Provided 0 is cast to a pointer to void (main2:82)
- Technically undefined if you want to be pedantic, but pragmatically it’s OK on the vast majority of platforms today (likely not all). And dereferencing 0 is fine on a system that has a valid memory page of 0 (e.g. in some kernels).

- quite common expression:
  a = malloc(..);
  if( !a )
    report_allocation_error(); (main2:87)
- Unless you already have silicon, it is almost certainly cheaper to change the architecture and ISA than to port the software for a non-zero null pointer representation. Even when you have the silicon already, the tradeoff might not be clear (depends how much custom software would you expect to write vs. reusing existing code). (main2:90)
- I know at least some systems have a non-0-representation NULL pointer, but none of the ones I care about. I think the standard should just get with the 21st century and declare that the target has a 0 NULL pointer and 2s-complement arithmetic... (main2:93)
- Very common to memset() a struct to 0 and assume the pointer members are NULL.

"Only sometimes" is probably only the deathstation 9000. (main2:94)
- It’s unsafe in some ancient systems, but those are dead and buried (except some of IBM’s mainframes, maybe). I’ve written memset(obj, 0, sizeof(*obj)) many times. (main2:96)
- Same remarks as 12. (main2:100)
- represented as, no; compared to, yes (main2:112)
- Memset is often used to initialize structs. (main2:115)
- Only on most platforms (main2:121)
- See answer 12 (main2:123)
- Any compiler or platform that doesn’t work this way is user hostile. (main2:126)
- I used to think it would be nuts for a modern system to use a non-zero NULL. However, for systems without page faults (e.g. embedded systems), it may likely make sense depending on the hardware’s memory mappings. (main2:129)
- More precisely, these two expressions must be identical

if (!p) ...
if (p == NULL) ...

- Yes, anything ever that interfaced with C++ (main2:138)
- NULL is supposed to be defined as (void*)0x00 not just straight 0. I’m sure the difference is negligible (spelling?) though it’s safe to say i haven’t thought of all the corner cases. (main2:141)
- A null pointer may not be zero on a given platform and runtime. For example, if you read byte-wise from a pointer that has been set to the null pointer constant, you will not necessarily see zero values. This is an unsafe assumption that may work on some systems but not others and so is non-portable. (main2:158)
- I do know that this is not guaranteed by the standard. (main2:160)
- Every system I’ve worked on is assumed to have null at 0. In fact, in my industry we don’t use NULL, we use 0.

BUT, sometimes 0 is a valid address. But we tend to ignore the concept of 0 as null in a smaller section of code that uses that address. Eq. One system had a small fast section of memory at 0. (main2:163)
- I have seen many instances of C code (most, but not all, 10 years old or older), treating 0 and NULL as interchangeable in pointer contexts. (main2:165)
- Pointer are initialized to 0 instead of NULL in a lot of code.
But I don’t know of any code which checks NULL pointers with comparison to 0 after having initialized it to NULL before. (main2:173)
- You’re supposed to use NULL.

Any code that contains

if (p) ...

Wednesday March 09, 2016
- Not in C++11. (freebsd:0)
- !ptr is such an awful, common idiom (freebsd:1)
- The C standard states that a NULL pointer is represented by (the constant?) 0. (freebsd:6)
- Note that the POSIX committee is currently discussing a requirement that a pointer value with all bits zero be treated as a null pointer (the requirement is specifically that memset() on a structure containing pointers initialize those pointers to nulls).
- I suspect that this part of the standard is there to support architectures which would now be considered "exotic". MIT Kerberos explicitly lists as a platform assumption that a representation of all zeros is a null pointer. (freebsd:8)
- There is no reason for the legacy allowance that the representation of a null pointer not be all zero bits. The choice of representation for a null pointer is arbitrary and need not match any hardware characteristics (except possibly generating exceptions, but compatibility with code that uses memset/0 is more valuable than such exceptions, in my opinion).
- Memset 0 to zero a struct. (gcc:3)
- I think most code assumes it’s zero even if the standard doesn’t mandate it (gcc:4)
- Plenty of code that uses memset(&x, 0, sizeof(x)); to clear the fields of a struct. (google:0)
- I’ve seen systems where NULL is just #define NULL 0 (google:1)
- something like NULL + pointer would be crazy (google:3)
- the standard says not to rely on it, but on most modern architectures it is true. (google:5)
- Stroustrup long ago advocated just using "0" rather than "NULL", and as a result I see lots of code that uses "0" when it means "null pointer". (google:7)
- I can theoretically envision a world where null pointers are stored with a special nonzero bit pattern, but since the C specification requires that 0 be treated as a null pointer, the compiler would have to do some nutty tricks to handle it.

I selected "there might well be" but I’m completely sure that there’s some code out there somewhere that relies on this being true in a way that would break if the compiler munged all null pointer constants to some other value. I can even guess what the code is doing (probably evaluating a pointer that’s been stored in an int, but possibly looking at a serialized representation). I just don’t know what software package is actually doing so. (google:11)
- memset() a struct to 0. Lots and lots of code. (google:14)
- old C code. (google:20)
- I frequently see zero and NULL used interchangeably, particularly in C where nulpnt is not a thing, and NULL itself is usually defined as 0 or ((void*)0). (google:21)
- Routinely see a literal 0 used in place of NULL. (google:22)
- Pointers need not be plain integers (google:24)
- Defacto? Yes; I have no idea about the standard (google:26)
- NULL is defined to compare with zero, not to be zero

There were several machines before unix that allowed zero as a normal memory location.
- This is, IIRC, the behavior defined in the spec. I’ve gotten into an argument about this one in the past. (google:29)
- It’s usually the de facto standard, even if it isn’t ‘officially’ the case. (google:31)
- The compiler might well use any invalid address as its bit representation of null. (google:32)
- If you mean a constant 0 in the textual program, then yes, and in C++ this is "normal" (at least until nullptr came to be), but compilers will probably warn and I find it confusing to use NULL, 0, and ‘\0’ with a different type on the LHS.

If you mean the actual value stored in the pointer variable (what you would get
from memcpy() into a char buffer) then no, you can’t do that in theory, but I have never used a system where it wasn’t 0. (google:34)
− Most real-world platforms these days use 0 as a null pointer representation, but some exotic systems don’t. (google:36)
− Lots of real-world code relies on being able to memset a struct to 0, and then assume all members are "zeroed" "properly". Typically they then expect float/double values to be 0.0, integers to be 0 and pointers to be NULL.

On most platforms this is actually true. Doesn’t change the fact that it’s not a requirement, and in fact it might make sense to use another representation than 0 for NULL when dealing with kernel mode code (on some platforms some global tables are near or at address zero). (google:37)
− I don’t think NULL == 0 is in the standard, but it’s universally true as far as I know. (google:38)
− For instance, nil (in Objective-C) is defined as (void *)0. (google:41)
− Lots of code assumes that NULL is zero, but on systems without virtual memory zero is a perfectly valid address, and why waste the space? On x86 zero is the interrupt descriptor table entry for INT 0. (google:42)
− See above. I need to rely on the fact that any not explicitly initialized global pointer is NULL (even if it is declared as a char array that was later aliased as a struct containing pointers). I need to rely on the fact that pointers in calloc()ed structures are NULL. (google:43)
− Yes (for what?): Using memset to zero-initialize structs that have pointer members. (google:45)
− Quite often code assumes that some area memset to all 0 will contains null, integer 0s etc etc etc. (google:46)
− Section 4.1.5 of the C Standard states that NULL "expands to an implementation-defined null pointer constant"

Also, http://c-faq.com/null/machexamp.html (google:47)
− Lots of code uses memset to clear out arbitrary data structures and assumes that it sets pointers to null. (libc:1)
− I see frequently pointer checks against 0 instead of NULL. (libc:2)
− void*p;
memset(&p, 0, sizeof p); // creates a null pointer

this is commonly relied on when initializing pointer arrays or structs with memset, posix plans to require this:

http://austingroupbugs.net/view.php?id=940 (libc:3)
− See above. (libc:4)
− Every C++-ported-to-C code ever. (libc:5)
− Almost everything relies on memset(s, 0, sizeof(s)) producing a structure whose pointer members have been set to NULL. (libc:6)
− It’s not that unusual to see a bug where NULL is accidentally returned from a function which has a return type of int, where zero means success. Oops. (libc:7)
− This is something that you can’t assume, but in practice, most platforms with a non-zero null representation aren’t worth supporting. (llvm:2)
− This is an odd question. The notion of NULL does not exist in C. You have a bunch of bits. Their value can be either one or zero. NULL is an abstraction from SQL, the idea that data has not been entered. The language is built around nulls but there is no such thing. You could add a bit to a number and set it to one if the other thing is NULL. But nothing would have a value of null. (llvm:3)
− You are going to find all sorts of crappy code that casts pointers to an integer type and compare the integer type to zero. They shouldn’t, but...

Casting pointers to integers is somewhat pervasive in Unix kernel code (check out the system 5x DKI apis -- every callback takes an integer parameter that is almost universally use to hold pointers). (llvm:5)
− It’s pretty common to see null-pointer checks that are simply:
if (p) { /* This is a non-null pointer ... */ } (llvm:7)
− It is very tempting to use all-bits-zero as the representation of null point...
ers because it means that all static pointer variable without initializers can be reserved from segments containing bytes initialized to zero, as found under the name BSS: http://en.wikipedia.org/wiki/.bss#BSS_in_C

The standard does not consider such implementation details. One would like to think that whether all-bits-zero represents a null pointer ought to be implementation-defined but the standard is not usually that helpful.

POSIX may add constraints about the representation of null pointers, but I am not familiar enough with it to tell. (regehr-blog:0)
- If calloc/memset stopped working to initialize data then it’d break a lot of programs. (regehr-blog:1)
- The non-zeroness of NULL pointers is a well-known legend, but is both a rare hardware issue, and ignorable if your code is well-behaved. (regehr-blog:7)
- 6.3.2.3p3 certainly suggests it. (regehr-blog:9)
- I don’t know if there are still any embedded chips whose compilers treat a non-zero value as NULL. I do know of code that relies on memset(memory_container_pointers, ..., 0) making pointers NULL. (regehr-blog:12)
- I can trust that assignment (=), equality test (==), and non equality test (!=) will treat 0 as indistinguishable from the null pointer value. I cannot trust that to be true for other operators. (regehr-blog:14)
- I remember reading that the standard doesn’t require it. (regehr-blog:15)
- Can be assumed but the standard does not force though IIRC, only the conversion of a 0 (a zero integer value) must be convertible to the null-pointer. I think this is the source of this confusion.

T* p;
if (p == 0) // ok
  if ( (intptr_t)p == 0) // arguable
    (regehr-blog:18)
- an initialisation of struct with help of as memset(&struct,0,sizeof(struct)) if the struct contains pointer fields (regehr-blog:21)
- I have mixed feelings about about this scenario and the previous. Requiring NULL to be equal to (void*)0 is the best and obvious choice, and should be mandated in the standard. On the other hand, there are very very very few occasions where relying on that fact isn’t a terrible idea – to say nothing of relying on compiler and system values for null, it also makes assumptions about memory space and layout that may be bad. (regehr-blog:24)
- Anybody who memsets a struct with zero bytes depends on this, if there are pointers in the struct.

There might be exotic C implementations where it isn’t safe, but every "normal" C implementation I’ve ever seen uses all-zero-bits to represent null pointers, and there is a LOT of code out there that assumes this is the case. (regehr-blog:26)
- Works on common architectures, where NULL is represented as 0, but not on some niche architectures with other representations.

Used when calling memset(&struct, 0, sizeof(struct)) & similar techniques. (x11:0)
- BSD kernel code passes 0 all the time when it means NULL.

C does define 0 as being special and equivalent to a NULL pointer. (x11:3)
- There is no real platform where null is not all-bits-0, and there never will be one either, except for specifically linting platforms like the DS9000.

The world is full of code that assumes it can calloc, or bzero, or whatever, and get null pointers back. In many cases, such as getting memory that’s known to already be zeroed, initializing explicitly incurs extra unnecessary overhead which people don’t want to pay for the sake of this piece of pedanticism.

Because of this, nobody will ever make a non-linting platform that has a different representation of null; there’s nothing to be gained from it and a lot of code will break.
I would even go so far as to say it’s not worth flagging this in a program analyzer, even at maximum pedanticity setting.

Note that the same concern applies for floating point representations, and there it’s not as cut and dried... (x11:5)
- !ptr (x11:6)
- Lots of code tests for NULL pointers using code such as:

```c
if ( !ptr )
  blah;
```

- Frequently new structures will be initialized to zero -- either with bzero or, more likely, allocated with zmalloc. It is assumed that the pointers inside the struct are now initialized to null. (xen:2)
- There is far too much code which assumes that the NULL pointer has the value 0. This is required under POSIX and can reasonably be relied upon, but there are architectures where it is very definitely not the case.

A very confusing matter is that in C "if ( ptr == 0 )" is a spec-permitted way of checking for the NULL pointer, even if the representation of the NULL pointer is not 0. (xen:3)

---

[14/15] Overlarge representation reads

Can one read the byte representation of a struct as aligned words without regard for the fact that its extent might not include all of the last word?

<table>
<thead>
<tr>
<th>Will that work in normal C compilers?</th>
<th>107 (33%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>81 (25%)</td>
</tr>
<tr>
<td>only sometimes</td>
<td>44 (13%)</td>
</tr>
<tr>
<td>no</td>
<td>47 (14%)</td>
</tr>
<tr>
<td>I don’t know what the question is asking</td>
<td>36 (11%)</td>
</tr>
<tr>
<td>no response</td>
<td>8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Do you know of real code that relies on it?</th>
<th>40 (13%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes, but it shouldn’t</td>
<td>39 (13%)</td>
</tr>
<tr>
<td>no, but there might well be</td>
<td>103 (35%)</td>
</tr>
<tr>
<td>no, that would be crazy</td>
<td>42 (14%)</td>
</tr>
<tr>
<td>I don’t know</td>
<td>67 (23%)</td>
</tr>
<tr>
<td>no response</td>
<td>32</td>
</tr>
</tbody>
</table>

Comment
- If the reads are word aligned then you can’t fall off the end of a page (main2.2:0)
- reading behind the last word can lead to segfault. Practically when reading aligned, that will hardly be the case but depending on hardware architecture... (main2.2:3)
- The last 1-3 bytes may be out of addressable memory which would result in an access violation at runtime (main2.2:5)
- There is an issue with aliasing rules in this case.

I read byte by byte with the believe that it is so widely used pattern that any sane compiler should see it and optimize accordingly (unless -O0). (main2.2:6)
- That sounds like UB to me (main2.2:7)
- All architectures I know of, do not offer protection more fine-grained than the largest load/store width of the processor. However, I can imagine an architecture that does support this, and constrains the compiler not to make such assumptions (beyond some load/store width). (main2.2:8)
- You can’t read off the end of a memory allocation.

I would expect to be able to read a struct using aligned words as long as I rounded down the number of words in the struct and read the remainder as bytes.
Hand-optimized code relies on it. This opens a whole can of worms with respect to endian issues and other things but if you know what you’re doing you need to be allowed to do this.

- reading as "aligned words" is something that the standard doesn’t allow at all, so this will be undefined behavior anyway.
- That’d probably page fault eventually. Broken in theory *and* in practice.
- I assume that the struct has the correct alignment, otherwise you’ll get a segfault.
- Do read as words, but only to exact extent.
- Compilers sometimes do that, but programmers should avoid doing it because it can easily introduce races (or reading uninitialized memory) and is generally undefined.
- The C version of strcmp() in FreeBSD is a good example.
- You need to be very careful doing that if you’re potentially near a page boundary.
- Valgrind will (rightly) complain though.
- I would expect this to be safe on all general purpose processors, but possibly bogus on special cores (e.g., shards of a much larger processor such as a GPU, DSPs, etc.), as long as the word being read is the platform-native word size (that is, not a 64-bit double on a 32-bit system, for example).
- Lots of code assumes that if you can read any part of a word, you can read the full word. It won’t always use the bits that aren’t valid, but some crazy code does.

Often you’d see this expressed as a variation on a theme of using bcopy where you might see a length computed by &a[1] − &a[0] rather than sizeof(*a) or sizeof(a[0]).

- There’s no guarantee a struct will be padded to word boundary (indeed struct { char a; } probably won’t be), so such a struct might legally start anywhere in memory and so it’s not clear how to go from aligned word-sized reads to the struct itself.
- That might generate a page fault if there is no memory associated with that final address.
- If I understand the question correctly it won’t work on little endian archs.
- Data might be stored in an unexpected format (like a float). It might be valid, but probably unexpected results, and would be machine dependent.
- The question doesn’t make sense. If you read the byte representation of a struct as aligned words, your result size is a multiple of the word size, which is not necessarily the same as the size of the struct. Thus the "byte representation" you have can never be the same as "the byte representation of the struct" because they are different sizes.
- If sizeof(struct foo) includes it, the padding can be read and would have been written.
- I’m not sure what the question is asking.

What do you want out of n?

1. One byte of n will be x.c, but I’m not sure it’s 100% defined which one. (And it’s possible that the compiler is doing 32-bit alignment and neither bit is, but that would generally be dumb.)
2. It is very unlikely that the read will segfault due to the padding bytes of x.
falling outside the page that the data bytes are on. I suppose it could happen if the padding bytes were inconsistent with the page size: e.g., pages were 1024 bytes but somehow the struct got padded to 1025. I can’t think of a non-stupid scenario here. (main2:55)

- I’ve seen code that does this as part of making efficient copies. I try to get people who do that, to use memcpy instead. (main2:59)
- Seems legit. memcpy + sizeof does this all the time. If that is illegal, we’d have some serious problems. (main2:61)
- In the picture processing code, by a mistake. (main2:64)
- Gotta be careful with that. Behavior that works fine on Linux can get you a bus error on Solaris. (main2:66)
- Type-based aliasing optimizations will break such code. (main2:69)
- While it’s undefined behavior, I think this might usually work. Not something I’d want to rely on though. All these questions about what modern "normal" compilers might do on undefined behavior is probably safest answered "only sometimes", but I feel like I’m copping out. (main2:70)
- Result will depend on architecture byte order. (main2:73)
- Incidentally, LLVM will do this to stack accesses in its optimizer. (main2:74)
- That memory might not be allocated. (main2:78)
- Don’t know this one. But my guess is that it could be problematic. (main2:80)
- I’m not sure how you read a struct as aligned words without reading the entire struct; a code example here would be great for dummies like me. (main2:86)
- I think you’ll get a warning for reading out of bounds (at least using valgrind). (main2:88)
- I’m assuming the question talks about reading past the extent of the struct when the allocator happens to round up the allocation request.

No this won’t work on normal compilers, if Clang with Address Sanitizer enabled counts as a normal compiler.

The code shouldn’t rely on it because it doesn’t really know if that memory is available as a part of this object. Or maybe I’m saying it just because relying on it breaks a useful tool (ASan). (main2:90)

- Depends on packed attribute I would expect. (main2:91)
- Libc memcpy is the usual offender here, though that’s often native assembly. Overreading is something I would avoid not for the sake of the compiler but because it will cause analysis tools like valgrind to complain. (main2:93)
- Depends a lot on the required alignment of the struct. If it has word-aligned members then the size of the struct will be a multiple of the word size so you will be OK.

Also depends on lack of trap representations in the type you are using to do the reads :-). (main2:94)

- If the struct has alignment requirements, sizeof() must be a multiple of the alignment, making the question inapplicable.

If the struct does not have alignment requirements, the rules say undefined behavior, but it works in everything I’ve seen. I’m sure strlen() is implemented exactly that way on most platforms (though that one is usually implemented in assembly). (main2:96)

- The struct might not be word-aligned, depending on the target ABI. (main2:103)
- It’s probably undefined; my biggest worry would be that C doesn’t restrict the granularity of memory protection, so reading even an aligned word past the end of a struct could cause a fault on some odd architecture. (main2:104)
- Well, you can read it, but like before, due to alignment issues it might crash at runtime. (main2:107)
- You would probably get away with this on almost all contemporary systems, but someone would try to run it on something with segment protection and then you...
would be in trouble. (main2:111)
- This sounds like undefined behavior (main2:121)
- I think it’s true in malloced memory and arrays, but probably is not true for single struct variables. (main2:123)
- I can’t actually think of a situation where this wouldn’t work just fine. (main2:129)
- question unclear, maybe because I’m not a native speaker (main2:133)
- no guarantees that hardware will not read the whole word, because of cache lines, etc. (main2:134)
- Use word copy for a byte array is potentially faster (i.e. 4 time less loops).

A lazy programmer would just the extra bytes (0 to 3 or 7 bytes) anyway. (main2:144)

- If padding is set to less than sizeof(word) e.g. none, it can generate an access violation.Malloc tends to collect small data allocations in large contiguous blocks. So out-of-bounds reads may often quietly leak part of some other data without generating an error.

ABIs typically specify at least sizeof(word) alignment. Legacy data have 16/32bit padding - less than the common 64bit word. Internal data can use any alignment. File formats commonly use no padding. (main2:147)
- It is possible due to how allocators align memory on (typically, now) 8 byte boundaries, but if reading packed streams, this is incredibly unsafe as it may cross a page boundary where there is no memory backing the address. (main2:148)
- All cases I have seen are one of:
  1. Is a latent bug
  2. Was a latent bug that was fixed by either padding the malloc or the struct
  3. struct is known to be embedded in another struct and so the reads are known to not be off the end of the stack or malloc (main2:154)
- This is a bit more sketchy. I would tell the person filing a patch to do this so to go back and try again, despite not knowing whether it’s spec or not off hand. But I would not be the least bit surprised to see it in production code. (main2:156)
- I’m not sure I understand the question. Is the idea that if you read a struct, say, using 64-byte reads at 64-byte aligned value, you might read past the defined end of the struct? I would say that reading past the end of the struct this way is undefined behavior, or should be. (main2:158)
- You would have to disable strict aliasing for this to work. (main2:160)
- Every system I’ve worked on has a malloc() that allocates at an alignment of 16.

But since we use our own custom allocators and are very explicit about alignment requirements we assume we can access up to that alignment. Eg. Doing string operations a word at a time. Also SIMD array access often processes more than it needs to. (main2:163)
- This is essentially the same as reading the value of an uninitialized variable in the case where the struct ends before the next word ends. (main2:165)
- I understand the question as: "the structure might end inside last word read".

To my mind, it works, but of course, the rest of the last word is undefined. (main2:173)
- What’s a "word"?

As long as it’s aligned, it won’t hit page boundaries.

More interesting question: 0-length struct.

(main2:176)
- Won’t generally work, because the last word will include uninitialized data past the extent of the struct.

Especially won’t work with __packed__ structs. (freebsd:0)
- I presume this is talking about a struct that is word aligned but may not be an integral number of words in length. If it’s not word aligned, then accessing it as word aligned would be machine dependent.
A segmented architecture (as allowed in ANSI C) could well have a segment ending at the last valid byte in the struct. Accessing beyond the end of the struct would generate a runtime exception – I don’t know if the compiler could treat this as undefined.

OTOH, some RISC architectures don’t allow accessing memory with a granularity any finer than a word – so forcing a C programmer to jump through hoops to avoid doing so would seem counter-productive.

Definitely, for doing block comparison, reading in words and then masking the last partial word is probably easier than reading the last part of the struct as bytes.

(freebsd:6) I would not expect the compiler to go out of its way to disallow natural (i.e., word-based) accesses. (freebsd:8)

(freebsd:6) If nothing else it requires the compiler to support something like GCC’s __attribute__((__may_alias__)); otherwise the read is undefined already due to aliasing violations.

(gcc:2) If there is code that relies on it, it’s probably buggy (gcc:4)

(gcc:2) Probably not very useful considering that the extra padding at the end would contain garbage. (google:0)

(google:1) Similar to my response above about padding but depending on the architecture you might have additional memory packed in against the tail of the struct.

(google:1) Memory sanitizers will hate you (and likely yell at you), but in general compilers and memory allocation routines guarantee you any object is fully addressable on full word sizes. (google:4)

(google:4) used for serialization. This generally works, but it might contain garbage in the last word and one needs to be conscious about byte order etc. (google:5)

(google:5) hash computation has a tendency to do this (google:7)

(google:7) This will work if you know that the structure is already word-aligned and implicitly padded, which I THINK (but I’m not sure) is the default case. If it’s packed instead of padded, it’ll be flaky -- it'll work as long as there’s something else allocated immediately after it, but if you walk off the end into a deallocated region you’ll crash, which you might not notice for a long time. (google:11)

(google:11) As I recall, structure size, unless explicitly specified otherwise, will be padded to the size of its largest member. So as long as your structure contains words, you’re fine. However, the size of a structure with one byte as a member will still be one, and it will be densely packed in arrays and in memory except where followed by a larger type. So you’ll get data pollution and undefined behavior treating small structures, or structures containing small amounts of data, this way. (google:21)

(google:21) Endianness.

Even if it works, I do not trust it to keep working in future compilers.

(google:27) I believe I’ve encountered this in serialization code. (google:29)

(google:29) There could be side effects (breakpoints, memory mapped I/O, page violations) (google:30)

(google:30) Due to possible structure packing, I can imagine this not working under some circumstances. (google:32)

(google:32) This isn’t portable, but in most cases a structure would be padded to the word size, and even if it isn’t you could easily get lucky with the out-of-bound read. (google:34)

(google:34) valgrind/asan will not like this :)

(google:34) It is more useful to have auditing against use of uninitialized/undefined memory areas than being able to read a struct this way. Thus doing so is expected to cause warnings, and possibly even breakage.

(google:37) Also, a separate "char" variable might just be stored in that "padding" spot, depending on how the compiler works.
- My concern would be possibly incurring a page fault on the last word and hitting a protected page. I don’t think it’s possible as long as your "words" are, in fact, the system word size, since pages should be a multiple of that size, but it’s definitely not a good idea.  (google:38)
- The padding bytes between struct members and at the end will contain garbage, if they are present at all.  (google:40)
- I think structs can be optimized by the compiler such that this is no longer possible, but I’m not sure.  (google:41)
- It won’t blow up the hardware with an extra page fault, at the very least.  (google:42)
- Not quite sure what I should answer to some of these questions. Yes, I *can* do that and I think it should be legal, well-defined C (regardless of what the standards lawyers say). What it will do is cause a word aligned memory access to that location. Whether that is a good idea or not depends totally on the situation (e.g. if this is targeting a memory-mapped device that only works with byte accesses it may generate an abort), but not in ways that have to do with the language itself.  (google:43)
- Anything that would cause this to break (OS paging, etc.) is word aligned (at least) and so won’t be a problem.  (google:45)
- In some debug environments it might fail.  (google:46)
- It depends on alignment, whether or not the struct is packed, etc...  (google:47)
- it may be ok for some code (where assumption can be made about the alignment of page boundaries), but i don’t see a safe way to access a struct word by word.

memcpy etc string.h functions often use word-by-word strategy, but such code must use "__attribute__((__may_alias__))" or equivalent when casting char* to word*.  (libc:3)
- This may break on platforms with various alignment requirements, or may leak data from surrounding structs.  (libc:4)
- If by this you mean sizeof(s) % sizeof(word) != 0, then we can infer that alignof(s) < sizeof(word), and therefore the possibility exists of a pointer to s being close to the end of a page, and therefore reading by words could incorrectly cross a page boundary and segfault.

If, somehow, the programmer knows by other means that this cannot happen, then I would expect it to work with -fno-strict-aliasing.

I can imagine some variant of this assumption being required to write a decent C implementation of memcpy, for instance. Of course, that would care for reading aligned words and therefore avoid SEGV, and it would of course not write out more bytes than requested.  (libc:6)
- Load widening is a pretty standard transformation, so it is hard to allow this while forbidding the user from doing it as well.  (llvm:2)
- There notion of representation is an abstraction. In C you can always strip away all the abstractions and just deal with the bits. So you can take an address and pretend it points at anything you like including aligned words.  (llvm:3)
- Most of implementations of heaps and program loaders round up the sizes of things because they are afraid code will do this. Especially, now, because of the cache line behavior of real processors.

So, this is technically bad, but because of the underlying computer architecture (caches), it will, for all intents work.  (llvm:5)
- This mirrors the padding question above; structures will be aligned to at least 4 bytes on most architectures (for stack, .bss, and .data alignment; heap alignment is usually larger). The alignment creates padding bytes, and the padding should not be access uninitialized. (That would be undefined behavior.)  (llvm:7)
- optimized memory copy functions do this. They cause Valgrind to complain, among other execution environments designed to help developers find bugs. (regehr-blog:0)
- What do you mean by "extent?" As far as I know this word has no precise technical meaning.  (regehr-blog:4)
- We’re talking about a read outside the struct, i.e., (p >= s + sizeof(s)) which is always a Bad Idea and should be avoided like the plague. Struct padding/ali
Should work on normal machines assuming struct itself is aligned; contents of the last word that aren’t part of the struct’s extent and aren’t part of any other variable’s extent would, of course, have a stable but unpredictable value.

I believe valgrind traps this one. MSan might. I’m not sure if any other environment does. It’s not hard to envision a "stupid, but fast" memcpy equivalent that does it, but I think at that point one should probably drop into assembly anyway.

The question does not state "how" the bytes are read (it is legal via "char*" but illegal via any other type). If the question is whether reads past the end of the struct are allowed, then the answer is the same as question 1.

Assuming you’re OK with an unpredictable value for the last word.

The Sun strcmp (or maybe it was strcpy) implementation famously did this. But then, the C library is part of the implementation and Sun controlled that.

We don’t work with shared memory or restricted memory. I would expect it to work in any case I would see, but I don’t think that’s normal.

Care is required to make sure the read beyond the end of the struct doesn’t cross a memory page boundary and access unmapped memory (or memory with access protections that prevent you from reading it).

As a game developer, I have sometimes done this to efficiently read packed data into vector registers from the middle of a complex data structure. You might want to read, e.g. 3 bytes into a vector register, but the efficient read instructions might read 4 or 8 bytes.

We either pad the size of the allocation a bit to make sure the read won’t cross a page boundary onto unmapped memory, or we control the low-level memory allocation in such a way that it won’t place the allocated block right up against the end of a page. Another possibility is to align the read so the extra accessed memory is before the part you want, rather than after (if you can guarantee that there is some unrelated data before all of the data you want to read in this way, you might already know it won’t cross onto an unmapped page, and not need to pad it.)

Relies on the implementation details of most memory allocators to round up to at least word size, and OS memory management/MMU to map pages aligned on word boundaries (and usually much greater, such as 4k or more).

If it’s not defined as "packed" and you use the alignment of the member with largest alignment needs, it will work. You have to be careful of padding.

Depends on whether the struct is word aligned and the CPU allows unaligned reads.

I have no idea why anyone would do this, let alone why a compiler would care. Structures whose sizes are not word-aligned are problematic anyway (if you have an array of them, what happens to the alignment?) so I would expect that there wouldn’t often be an issue.

That said, one would like to be able to have e.g. 3-byte structures (such as for RGB values) without incurring padding bytes, and have them work, in which case reading them as aligned words is going to be problematic. But I have no idea why anyone using such structures would try accessing them as words.

I suppose technically one is only allowed to access representations using character types, so accessing as words can’t be expected to work.

This question was hard to understand.

If the allocation is not aligned on at least a word boundary (which is not guaranteed on architectures that allow unaligned accesses) then dereferencing beyond the end of a struct may walk over a page boundary.

A lot of these questions boil down to whether you are doing your own memory management, or using malloc(). If you’re letting malloc() do your memory management, then you shouldn’t assume anything except what malloc() promises: you shouldn’t assume, for instance, that you will be even able to read a byte past the official end of the struct, even if you "know" that it’s architecturally impossible.
le to cause a page fault as a result (since a word-aligned read can never cross a page boundary).

But if you’re doing something where you control the memory (i.e., you’re mapping pages or you’re reading a large data buffer in from a file) then you may know for other reasons that this bit of memory past the end of a struct is valid, and so it may be acceptable to do something like you’ve described here. (xen:2)
- I would expect to work in general, although I expect it is probably UB. (xen:3)

---

[15/15] Union type punning
When is type punning - writing one union member and then reading it as a different member, thereby reinterpreting its representation bytes - guaranteed to work (without confusing the compiler analysis and optimisation passes)?

Type punning:
- If the members are the same size (main2.2:0)
- never (main2.2:1)
- Not guaranteed at all (main2.2:3)
- When the member types are somehow compatibles. In practice (but not in principle), when one member type is a struct whose fields are the initial fields of the other member type. (main2.2:4)
- when the two types are related (main2.2:5)
- At least in cases where the union members are structs/fields with same start. E.g.

```c
struct SimpleX {
    uint32_t n;
    SomeDataPayload payload;
};

struct ComplexC {
    uint32_t n;
    SomeDataPayload* payload;  // array of multiple
};

union X {
    uint32_t n;
    struct SimpleX s;
    struct ComplexX c;
};
```

I strongly believe that accessing "n" via any union member is equivalent and interchangeable. (main2.2:6)
- No idea (main2.2:7)
- Between integers of known size and alignment and bytes, and compositions of this. For strings, need to know the character size (e.g. whether wide chars are used). For floats and other types, it’s possible to reply on representation by making the code dependent on the particular architecture / representation. (main2.2:8)
- It’s always guaranteed to work.

Type punning is dumb, though. It means I have to rely on the optimizer optimizing out the union. People were type-punning using pointer casts for a very long time before compilers started getting strict with alias analysis.

When I need to type-pun I use __attribute__((__may_alias__)). (main2.2:9)
- In C89, it’s UB. In C99 it’s allowed (with the comment that the result is implementation-defined, but it works in practice as expected). I don’t know of C11, but I assume this didn’t change. (main2.2:10)
- one is part if the other (main2.2:11)
- Works between integers of same size but different signed-ness. May work with pointers to different types. (main2.2:15)
--fno-strict-aliasing (main2.2:17)
- Yes, this needs to work. Among other things, math libraries sometimes need to do this to tease apart different fields inside floating point numbers. (main2.2:18)
- this is not clear from the standard; the gcc documentation takes some stance on this, but I don’t know whether I can reproduce that faithfully here (main2.0)
- Never? Use memcpy (main2.1)
- When given appropriate flags. Generally never. (main2.3)
- It’s valid so long as the stored value of an object may be accessed through an lvalue of struct/union type that includes one of the types amongst its members. (main2.4)
- The two members must be of the same size. (main2.5)
- It better work. My company currently uses the following idiom extensively:

  union{
    U32 Fault;
    struct {
      U32 Fault1 :1;
      U32 FaultCode2 :2;
      U32 Fault3 :1;
      ...
    }
  }

- Can’t remember, I try very hard to not need to do that. I think it’s fine most of the time, but would check first. (main2.7)
- I don’t know, because I’m too confused between the differences in the C, C++, and OpenCL standards. I thus just use memcpy. It’s probably working when the types involved are all integers of the same size (signed or unsigned), or if one of the types is unsigned char. Or any type of char. (main2.8)
- I do this via the union trick (not OK by standard) to do SIMD. (main2.9)
- Never, but compilers agree that doing so through memcpy is ok, won’t be broken, and won’t have a performance cost. (main2.10)
- According to the standard never; in practice always. (main2.12)
- never.

But one would expect it to work for normal types if performed in different translation units. (main2.16)
- In a C11 compiler. In C99 this produces an unspecified result but not undefined behaviour. (main2.18)
- If I recall the standard correctly, it is never okay to read the union as an other type. However, I’ve seen plenty of compilers emit the expected code. I also have seen xlc assume that the read could give back garbage, until we added some tricks to force the compiler to not optimize it away. (main2.20)
- Always, it is deterministic. (main2.21)
- Always? I suppose there might be problems if the two members don’t have the same alignment, or if the written member is smaller than the read one. (main2.22)
- As long as all accesses are via the union, and not, say, by taking separate pointers to the union’s fields. (main2.23)
- WAT (main2.24)
- When you cross your fingers and hope the compiler doesn’t warn (main2.26)
- I consider this generally dangerous and would use it only in controlled and tested circumstances. I would expect it to be uniformly safe if the storage and read are separated in code space such that static analysis can’t tell that the two accesses are always the same value. How separated that has to be varies widely in practice, even with superficially similar code under the same compiler!

I would LIKE to say this is always safe. Up until a few years ago, I would have expected it to be. I am no longer so confident. (main2.27)
- I would like to imagine that “direct” access (write u.x, read u.y) would always work (assuming the bits represent a valid value for u.y). But I can also imagine that “casting through a union” would not work: ((union my_union_type*)&value)->another_member.

(main2.28)
- Yes. That’s the only guaranteed way to do aliases. (main2.29)
Should work all the time. (main2:30)
I think this will work provided one does not attempt to dereference beyond the end of the initial member, in which case the result is undefined. (main2:32)
No. (main2:33)
No idea when this is guaranteed to work! (main2:35)
The overall (byte level) lengths are equal. (main2:36)
Type punning always works. The compiler knows very well which fields in a union have what offsets so it knows what writes to one union impact which fields in another member of the union. It should not be confused. (main2:37)
It should always work, as this is very common trick. (main2:38)
Only 1 member? (main2:40)
As far as I know (main2:42)
When the union member sizes are equal (and that’s a guess). In practice, I wouldn’t do it quite happily for non-equal sizes. (main2:44)
It is legal, but may prevent the compiler from performing some optimizations (main2:45)
When the member written is written via the union, i.e. as: "u.a = value;" and the member read is also read via the union, i.e. as "var = u.b;". (main2:46)
Always (main2:48)
When you want people to buy your compiler :-). (main2:49)
At least when doing it sufficiently "in the compiler’s face", i.e. making the accesses through the union (and not a pointer to a field thereof), or when compiling with no-strict-aliasing. (main2:50)
No, but I believe pretty much all compilers go to some lengths to ensure it does, because reliance on this is common. (main2:51)
I’d expect it to work if the union members have the same types in the same order. (main2:53)
I believe it works where the two members are structs that share a common prefix, and you’re only reading the prefix fields; I know of real code that does that. I suspect it often works in other cases, and wouldn’t be surprised to learn of code that relies on that, but I don’t know for sure. (main2:54)
Sheesh, don’t do that, use a cast if you must do that at all.

I don’t think it’s *guaranteed* to work ever, by the spec.

It will generally work if the from- and to-types are the same size. But the compiler *could* align them differently. You’re probably pretty safe type-punning one pointer type to another, but, seriously, there is no excuse for doing it.

It will never pass Google code review. Ever. (main2:55)
In theory (at least for C++) only for standard layout types with a common initial sequence when reading member of that common initial sequence. However, the practice is so widespread all compilers I know of generally support it. (main2:56)
When the union is declared volatile? When memcpy is used? Honestly, I don’t know – I think it’s always undefined because -fstrict-aliasing makes assumptions that such things don’t happen. This is an area where I think the standards folks made the wrong decision; use of "union" is rare already and using unions as a way to do type punning without resorting to memcpy is a major use of unions. (main2:59)
When all the types involved are POD and of the same sizeof(), I think (main2:60)
I have no idea. Erring on the side of safety, I’ll say this is fine if one of the types is char[] and has the same size as the other member. I’d just use memcpy though. (main2:61)
I believe this always works on compilers that I use. I think it is guaranteed by those compilers. (main2:62)
Unfortunately it is not guaranteed, meaning there is no way to safely perform "unsigned int" <-> "float" representation changes. People do all sorts of stupid trickery like memcpy to satisfy unbearable and irrational requirements.

We need a bit_cast. (main2:64)
When the types are the same size (main2:65)
When the members are the same size and there are no alignment concerns. (main2:66)
n2:66)
- always (main2:68)
- GCC and Clang try to allow it when it’s sufficiently obvious that you’re doi
ng type punning (for instance, when you’re directly accessing a block-scope unio
n variable). GCC documents this, Clang does not (and only really does it for GCC
compatibility). (main2:69)
- I think it should be okay as long as the one you write to is the same or lar
ger than the different member you read from, as the values of padding bits are u
nspecified when writing to a smaller object than the one you’re going to read fr
om.

(main2:70)
- types are compatible (main2:71)
- No idea. I would assume, if interested specifically with how it affects opti
mizations, that it could be compiler specific. (main2:72)
- no idea. (main2:73)
- Per the standard? Never. The conforming way to do this is with memcpy to a l
ocal, and the compiler is plenty smart enough to not actually emit the memcpy or
the local.

GCC’s documentation claims that they support this as long as you’ve declared the
union in advance. This is pretty scary because it means lexically in advance. S
o two identical function bodies before and after an unrelated declaration introd
ucing a union may change the generated code for the two functions. In practice t
ese unions go into header files and come before the rest of your code, so peopl
e tend not to notice.
(main2:74)
- union {
  uint32_t u32;
  uint16_t u16b[2];
  uint8_t u8b[4];
} x;

x.u32 = 323232232
(main2:75)
- always (main2:76)
- Must cast to char* first. (main2:77)
- Never? (main2:78)
- I don’t think it’s ever guaranteed to work, unless perhaps one of the union
members is a char array. (main2:79)
- You read it as an array of bytes.

Or use a compiler like gcc.
Or compile under C99 or C11 standard. (main2:80)
- The standard guarantees that the read attempt will take place, but it is und
defined for casted pointers. (main2:81)
- When alignment, endianness and structure allocation are respected, with full
allocation (no padding). Cannot think of a portable solution across different
targets/compilers. (main2:82)
- All the time. (main2:83)
- Not sure on details (main2:84)
- when type-based aliasing is disabled (main2:85)
- Loosely speaking (perhaps accurately), when strict aliasing rules aren’t vio
lated.
(main2:86)
- Only when compiler explicitly allows it? (main2:89)
- I’d expect it to work when used as the exact type punning idiom -- declare a
union, store one member, load another. There’s just too much code out there th
at does this, since someone on the Internet said that it’s legal according to th
e C standard, and other people continue repeating that.

On many compilers union type punning would always work, regardless how the code
is organized. (main2:90)
- not sure (main2:91)
- only when one of the types is a char type. otherwise, never guaranteed to wo
rk. (main2:92)
- This is firmly in the "if I need to know the answer to this question I will
always look it up" category. I know there are beartraps here... (main2:93)
− You are allowed to pun the prefixes of structure types when the struct members in the prefix have the same types.

Unsigned char [] and other types is probably OK.

Otherwise, you are getting into strict aliasing problems. (main2:94)
− It should always work? (main2:95)
− I have never seen that break. Optimizations are common, but they do exactly what I want and expect. It’s a common test for machine endianness. (main2:96)
− yes, it will work (main2:97)
− Same size and offset. Alignment shouldn’t matter, Union will pick the max (main2:98)
− I thought that this was never guaranteed to work. Perhaps adding volatile in the right places would make it work (but renders the optimiser useless)? (main2:99)
− AFAIK the only guarantees are for structs with common initial sequences and usual permitted-aliasing rules. This is rather sad since it precludes (for instance) some natural ways to get at the bit pattern of a floating point value. As above I assume that compilers will exploit the freedom the spec gives them. (main2:100)
− Basic types of the same size (e.g. float and uint32_t).
− I expect this to work for a union of A and B in roughly the same cases where casting a pointer from A* to B* and dereferencing would work, and in particular:
− Accessing common initial members of two different struct types
− Accessing an object of any type as an array of unsigned char
− Never, so far as I know. (main2:104)
− Only when it is part of a sequence of initial elements with similar type. (main2:106)
− As long as the alignment matches, that’s entirely possible. (main2:107)
− If the members are the same size it’s generally guaranteed by the compiler to work (but AFAIK not by the C standard, at least not C89). (main2:108)
− Guaranteed to work from C99 and later. (main2:109)
− Per the standard, only when punning between other types and (unsigned) char, but in practice I’ve never had problems punning between float and int32_t, which is good for some numerical algorithms, as it usually avoids endian issues. This is well supported by most modern floating point instruction sets, where the same registers can also do most integer operations. I expect that emscripten would break here, but I’ve not tried it yet. (main2:111)
− When the alignment and endianness of the values are the same. e.g. a union that is both a char array and floats must be the same endianness of the given system. (main2:112)
− when they have the same size (main2:113)
− Never. (main2:115)
− At least when the two members have the same type.

Also when the two members have the same size.

I’m not sure that it does not confuse the compiler but it probably works when the size of the read member is inferior to the size of the written member. (main2:117)
− pfft what is a ujniuon (main2:119)
− I think its probably guaranteed when size of(type1) == size of(type2), and they are both the same type of value (i.e., int32_t and int16_t[2]) (main2:121)
− I know where to read about it if I needed to know, but I am unclear on the details. (main2:123)
− It’s suppose to. (main2:124)
− In practice? Basically always. Not doing this would break a great deal of code.

Per the spec? I think only if they have the same initial members. (main2:126)
- It will read something but not guaranteed to be consistent. (main2:127)
  - This will always "work" as far as I know. (main2:129)
  - Never (main2:130)
  - I think people generally expect this to work. I understand it’s never guaranteed to work - so I avoid it personally. (main2:131)
  - when tested properly (main2:133)
  - only on big-XOR little-endian machines. I forget which because I avoid it. (main2:134)
  - Never (main2:135)
  - All the compilers I know of allow this, unless you explicitly set some magic flag (which in gcc isn’t even included in -fstrict-aliasing). (main2:137)
  - Never guaranteed to work, but, in my experience, always does. (main2:138)
  - Yes. Common use I have seen is converting between different integer types and using bitfields to access hardware registers. (main2:140)
  - I’ve done it before with a union with two members, of the types:

```c
(uint8_t *)
(uint16_t *)
```

I calloc’d the whole struct w/ zero beforehand so maybe i’m depending on that (in bad practice) though it seemed to work fine with me. (main2:141)
- yes, although it generally confuses the programmer. (main2:143)
- It will definitely depend on endianness and padding.

I know of deliberate use of that technique to swap endianness before transmitting or after receiving data. Definitely not portable, (yet efficient!)

- When the destination type has no invalid values. (main2:147)
  - always (main2:148)
  - Never (main2:149)
  - one of the union members is an unsigned char array (main2:154)
  - -fno-strict-aliasing and friends? I don’t know, this is an area with hazard lights that requires googling to get right every time. (main2:156)
  - Not sure. I know I’ve seen this in code, though. (main2:157)
  - I am not entirely clear on what the standard guarantees here. In general, I think it is best practice when using unions to use a value that distinguishes the runtime type, for example, build a struct type that contains an enumeration in dictating how you wrote the union, so that you can read it back later using the same member. It would not be undefined behavior to write data with one member and read it back with another, but I can’t comment on what it might do to the optimization. (main2:158)

- I think it’s only guaranteed to work with a variable declared of the union type? (That is, I’m not sure what happens if you use a pointer of some other type and then cast it into the union type.) If I wanted to write such code, I’d look up the safe way to do it. (main2:160)
  - Never (main2:161)
  - We only do this in structures in code where performance isn’t critical. We usually do this by casting pointers instead because we don’t trust the optimiser (we check the disassembly a lot in this industry and unions cause all sorts of optimisations to be disable around the union). (main2:163)
  - When the types have the same size—e.g., uint32_t and int32_t. (main2:165)
  - When the types take up the same amount of space (main2:166)
  - Always? (main2:168)
  - General rule: Only on the machine it has been tested (endianness), with the compiler which it has been tested (optimizations).

I personally expect it to work:
- between int and uint of same length,
- int and pointer of same length,
- int and enum of same length (when the representation of the enum is known)
- floating point types with char[] (bigger int types: endianness problem) (main2:173)
- When the field is accessed using the . or -> operator on a value or pointer of union type. Possibly in other situations. (main2:174)
  - yes, e.g. gcc specifically mentions this as a safe way around strict aliasin
Where are the radio buttons? (main2:176)
- No idea (main2:177)
- It will work as long as the two union members are of nearly the same type, like signed int and unsigned int. But it won’t necessarily work if the union members are of different sizes, or even if one is a float and the other a int32_t. (freebsd:0)
- When they would have equivalent representation? I try not to do this. (freebsd:1)
- From my understanding per the C spec, it is not allowed, but often used. It may be allowed if you throw in a char pointer cast to help defeat aliasing. Though it is often used, I’m no longer a fan of it. (freebsd:5)
- My understanding is that this is always valid. Definitely (eg) FreeBSD libm relies on it to break up floating point numbers. (freebsd:6)
- When writing to it character values that entirely span the size of the other type and were generated by reading character values from the same or another object of the same type (that was properly initialized). (In essence, the same guarantees as you have for, example, using fread()/fwrite() to transfer a value to external storage.) (freebsd:7)
- I believe this is only permitted for union members which share a common initial sequence (but people do it for cases which do not have such an initial sequence and expect it to work). The effective type is set at assignment and accesses from non-compatible types are not permitted. (freebsd:8)
- It’s not guaranteed to work in Standard C, but it is in GCC and (I understand) many other C compilers. (gcc:0)
- The GCC compiler explicitly guarantees that union type punning will work, as a language extension. I believe that clang follows suit, and I expect most other compilers do as well. The language does not permit it. (gcc:1)
- Always (modulo issues like trap representations). (gcc:2)
- When the initial elements of the union are the same types, or when one is char or unsigned char. (gcc:3)
- When the two elements are the same size? (gcc:4)
- In theory never; in practice maybe always? (google:0)
- When the two representations have the same size and alignment. (google:1)
- When the member is marked as volatile. (google:2)
- No idea (google:3)
- Not sure what the question is, I’d say this is fine for any POD. (google:4)
- According to the standard, it’s undefined, but gcc explicitly documents that it is allowed. (google:5)
- When you are reading the members of two structs/classes, who start out the same way. This is useful for, e.g. the small-string optimization - you want a single byte that determines how to read the rest of the structure. (google:7)
- I’m not confident on this one. I know a few cases where it’s NOT guaranteed to work, but I’m primarily a C++ programmer and I’m not deeply familiar with how smart the analysis and optimization passes are. My intuition says that it’ll surely work if all of the elements are non-pointer primitives, and that it’s safe if all of the elements are pointers of the same size (but at that point you could just be using void* instead of a union), but I don’t know what optimizations will be thwarted in doing so. (google:11)
- I don’t know when it is guaranteed to work, but I assume always. (google:13)
- Yes, for small values of "guaranteed". I’ve seen this used to construct NaNs. (google:14)
- I don’t know. (google:18)
- Without confusing the compiler analysis and optimization passes? No idea. In general? A union is the safest way to do this. I see a lot of code cast from double* to char* directly, without using a union. (google:21)
- All the time, afaik. That’s how you’re supposed to do it. (google:22)
- Yes, it should work. (google:23)
- Never, AFAIK (google:24)
- When the union members are the same size, it should be OK. (google:27)
- I do not know. I believe it will "work" in the sense of generally returning
the results the programmer was expecting, but I do not know how confused the compiler will get in the process. (google:29)
- Both are correctly aligned and both represent a legit value (google:30)
- If the type being read is smaller in size than the type written, it should work. If you try vice versa, the more-significant bits may confuse things. (google:31)
- I believe this works when the punning is done via a union, although not necessarily through pointer-based type punning. (google:32)
- I don’t believe it is guaranteed to work, though it does in practice with many compilers and GCC has an extension which specifically allows it in certain cases. (google:34)
- The rules are too complicated to remember --- it’s easier to change the compiler flags to make this work than to change the code. (google:36)
- I’d say this is at least supposed to work, given people need this a lot and it’s the only "legal alternative" to type casting pointers which breaks strict aliasing rules.
- Can’t say whether the standard requires it to work. (google:37)
- When the types are the same size. There may be problems with integer versus floating-point if the compiler is generating special floating-point instructions. (google:38)
- If the member used to read the contents of a union object is not the same as the member last used to store a value in the object, the appropriate part of the object representation of the value is reinterpreted as an object representation in the new type as described in C11-6.2.6 (a process sometimes called @M-^@M-^XAM-^@M-^Xtype punning@M-^@M-^YAM-^@M-^Y). This might be a trap representation. (google:39)
- My mental model for this is that it will work as long as the binary representation for all unioned types is well-defined. I understand that the C spec is loose enough that it allows integers which are not 2’s complement and non-IEEE754 floats/doubles, but I think such systems are rare now. So I would expect this to work for various integers and floats.

I would expect this to work for pointers if -fno-strict-aliasing is given. I’m not sure if there’s a difference in the spec with respect to pointers between casting and union type punning, but I would expect them to work the same.

I would not expect this to work for structs, since padding varies according to architecture, compiler, and version. (google:40)
- I don’t know. I tend to stay away from this kind of thing, so I’d need to look it up. My guess is that as long as the one type is the exact same size as the other type, it will work. (google:41)
- Always or you have a broken compiler. (google:42)
- Type punning via unions is always guaranteed to work. The only problem I’m aware of is type punning by directly casting a pointer to one object to another type and dereferencing it, which breaks strict aliasing rules unless one of those types is char. (google:43)
- I don’t know of any cases where that’s allowed. I guess there may be special cases that I don’t know about where it is allowed, but I’d have to look it up. (google:45)
- Conversion to characters (character arrays) will work always. Typically conversion to other type works. (google:46)
- If they have the same alignment (google:47)
- It should always work I think as long as C aliasing rules are followed, and unions should take care of this. (libc:0)
- Nobody really knows the answer to this one, I’m afraid. (libc:1)
- union type punning should be reliable.

Taking the address of the members of such union and passing the pointers around to access the underlying object should not be relied on.

e.g. various functions in libm cannot be implemented in c without it, various floating-point code assumes ieee representation (and same alignment, endianness for uint32_t and float) and then
float x;
//...
union {uint32_t i; float f;} u = {.f = x};
u.i = process(u.i);

is the idiomatic way to access or manipulate the bit representation of a float.

(note that complicated mixed u.i and u.f arithmetics often cause the compiler to generate suboptimal code: it can even generate spurious load-stores from fpu regs to memory and cpu regs. the result would be observably wrong if either u.i or u.f had trap representations, without such traps this can still cause order-of-magnitude slow-downs because of subnormal values are spuriously load-stored to fpu registers).

(libc:3)
  - I'm not sure, and try to avoid using this feature because of it. (libc:4)
  - For non-struct, scalar values. (libc:5)
  - When the compiler manual says it will work?

Although honestly this is such a useful extension to ISO C that it ought to just be standard by now. (libc:6)
  - When you have a function call boundary in the middle. That is, write a union member, then call into a function which reads the other member. (libc:7)
  - The standard's position is not clear to me, but as a practical matter I believe the load, store, and the intervening code must be in the same TU. (llvm:0)
  - I don't think it is ever guaranteed to work. However, in practice I think compilers attempt to make sure that works for simple local unions rather than having to use memcpy to do bit-wise type punning. I expect compilers to be able to see obvious type puns involving unions as obvious aliases even when type-based alias analysis might say the memory references should not be aliases. (llvm:1)
  - It isn't really guaranteed to work. It's provided as a best-effort loop hole for users of strict aliasing. It is very easy to confuse the compiler by taking the addresses of fields and moving the stores around into different functions until it can't see the "union-ness" of the original allocation anymore. (llvm:2)
  - Optimization is compiler dependent I guess. We use unions to discover if a process is little or big endian. It is reliable and should be immune to optimizations. (llvm:3)
  - no guarantees, punning int to float and back often made to work because many older codes use that to perform fast float->int tricks. (llvm:4)
  - When all of the fields before the member are the same type and the members being accessed are of the same type (same type would also include alignment extensions). (llvm:5)
  - Don't know for sure. (llvm:6)
  - Guaranteed to work at all times. ;P

In all honesty, union type punning has many parallels with typical data type casting, but can provide stricter rules around how the data may be accessed. A union of data types with different sizes must guarantee that accessing the largest of the data types always has a predictable result. A contrived example with expected portability problems;

union A {
  uint32_t x;
  char y;
};

union A a;
a.x = 256;
a.y = -1; /* Sets the first byte of the union to 0xFF; a.x == 511 on little endian platforms, or 0xFF000100 on big endian platforms. */ (llvm:7)
  - It was always allowed in C, but C99 TC1 and TC2 were unfortunately worded. C99 TC3 makes it clear that the intention was to allow type-punning through structs all along (footnote 82).

This is not allowed in C++, apparently, but a compiler is still free to document that it allows it. GCC does. (regehr-blog:0)
  - I think this changed in a recent version of C.
Without checking the standard, I think now it’s not UB, but you may get trap representations. (regehr-blog:1)
- if the read is through a union member of type unsigned char[LARGE_ENOUGH] (regehr-blog:2)
- This is UB

In practice it works in both GCC and clang (regehr-blog:3)
- It seems the current state of affairs is that, due to the infamous "type punning" footnote added to the standard, compilers special-case the direct access from a union member, but usual aliasing restrictions kick in as soon as the access is made indirect (e.g., through use of a pointer). (regehr-blog:4)
- err, always? (regehr-blog:6)
- I assume type-punning always works, mostly because I don’t understand why it wouldn’t. (regehr-blog:7)
- It usually works. (regehr-blog:8)
- In theory: when the type is punned through an array of unsigned chars.

In practice: it always works when punned through a union. It also always works when punned through horrible pointer casting, but that’s more fragile (if UB can ever be said to not be fragile). (regehr-blog:9)
I’d guess not, but I’ve never seen it fail. Maybe you just need a few "volatile"s if you’ve got an unhelpful compiler :s (regehr-blog:10)
- always. if someone’s doing it, it’s because they’re playing games with byte-level representations. (regehr-blog:11)
- As far as I know, union-based type punning is allowed only when one of the two types is an unsigned char [], or when reading a value from a field in the initial common subsequence of fields in the structs used for reading and writing.

(I think the second part is only true in C99, 90’s-vintage gcc, and later.) (regehr-blog:12)
- I don’t trust this to ever work.

I am in the process of refactoring code that relies on type punning.
I considered this to be a tricky & clever technique in my youth. (regehr-blog:14)
- It seems to be ok as long as the values are always accessed through the union. So no passing pointers to union members around. (regehr-blog:15)
- When the other member is an array of chars. (regehr-blog:16)
- I believe the implementations I typically work with explicitly allow this, so I’d expect the answer to be "always" for my own purposes, though I’m not sure if/how that would generalize to other implementations. (regehr-blog:17)
- Well, a quick read of the type-based aliasing rules seems to suggest that always (even if the members of the union have different sizes). Details are probably addressed elsewhere. (regehr-blog:18)
- It’s not guaranteed by the standard, but compilers allow it (e.g. it works in gcc ven with -fstrict-aliasing. Like using a cast to reinterpret a value as a different type, type punning via a union should always work. (regehr-blog:19)
- It is not. You can only access the union via the member which was most recently written. Because, IIRC, the C standard says so. There might be wiggle room for cases like this where the underlying types are actually the same, but I would not be certain:

typedef int blarg;

union u {
  int a;
  blarg b;
};

union u you;
you.a = 2;
printf("%d", you.b); (regehr-blog:20)
- It should work in any case, struct/union is just a bytes with named offset as a field (regehr-blog:21)
- No idea (regehr-blog:22)
- I don’t know. (regehr-blog:24)
Casting a pointer-to-A into a pointer-to-union type (or directly to a pointer-to-B) and then accessing it, is unsafe unless you use a compiler option to disable strict aliasing.

The strict aliasing rule means you can’t access the same storage "as two different types" (e.g. int and float), unless you use the union or one of the other exceptions e.g. character types. One safe way to type-pun, that actually is optimized well by at least some modern compilers, is to use memcpy. But if the compiler is not smart enough to optimize away the memcpy, it can end up hurting performance.

I hate the strict aliasing rules because it has the worst possible failure mode: you’ll write code that seems to work, and passes your tests, and you’ll check it in, and three months later someone will adjust some code or tweak some optimization settings and something will get inlined or optimized slightly differently, and suddenly the code will "misbehave". Like some other kinds of undefined behavior, strict-aliasing violations are a time-bomb waiting to happen. Despite the performance loss, I prefer to just disable strict-aliasing with a compiler switch, especially on a large-team project that contains low-level bit-twiddling code. (regehr-blog:26)

- Not sure about confusing the analysis (I reported inefficient code from that as gcc 2.2.2 bug), but we use this in one place, where we did not just recast the pointer to the other type and access that because the existing arrangement in memory may be wrong, and because of alignment concerns. In general we just cast the pointer and access the memory through that, though. (regehr-blog:28)
- should work all the time, otherwise a union becomes useless. (regehr-blog:29)
- When the accessed fields are read as char/char array. (regehr-blog:30)
- Yes (regehr-blog:31)
- When the union members have the same size & alignment requirements. (x11:0)
- It should work. It would be useful, for example, if you have a union with members an integer and a pointer, and you want to write it to a file or something. (x11:1)
- when the types aren’t pointers. (x11:3)
- This was explicitly legalized by C99 TC3. IIRC. And before that it was a widely used practice. So one has to suck it up.

It would be nice to have a way to tag this usage (or, alternatively, to tag unions that are really sum types) for program checking purposes, but it’s not trivial. (x11:5)
- with what result? (x11:6)
- This is commonly used for parsing bit fields in network headers. Clearly bit field ordering is compiler dependent and therefore the code is non-portable, but such code is commonly found in Windows drivers. (xen:0)
- No idea. :-) My instinct would be to use such a construct with a type and then a char or unsigned char. Accessing something as an int and then a float seems kind of crazy to me. :- (xen:2)
- I really have no idea because I have repeatedly got lost in that part of the spec.

I do not trust anything compiled without -fno-strict-aliasing (xen:3)

Other differences
If you know of other areas where the C used in practice differs from that of the ISO standard, or where compiler optimisation limits the behaviour you can rely
on, please list them.
Other differences:

- I’m sure there are loads.

The risky behaviours I rely on most are:

The ability in C++ to cast between pointer-to-member-functions, and call the member function even though its been cast to an undefined type:

class undefined;
class foo
{
public:
    int member_func();
};
typedef int (undefined::*mfunc)();
mfunc m = (mfunc)&foo::member_function;
foo p;
undefined* q = (undefined*)&p;
(q->*m)();

I think this is actually technically legal.

A C++ constructor leaves the memory its instantiated on alone, unless it actually sets values, e.g:

class foo
{
public:
    foo() {}
    virtual ~foo() {}
    int member;
};

struct i_know_too_much
{
    void* vtbl;
    int member;
};

// assume aliasing issues dealt with
i_know_too_much p;
p.member = 42;
foo* q = new (&p) foo();
assert( q->member == 42 );

Which is probably very naughty but it means I can serialize C++ objects as blocks of bytes and fix up the vtbls afterwards.

(main2.2:9)

- âM-^S quite often left-to-right function argument evaluation order is relied upon.
- âM-^S Pointer arithmetic is abused wildly and in creative ways. The most popular two examples are:
- âM-^S an attempt to rely on the contiguous allocation of multidimensional arrays, by traversing them using a single pointer to the first element:

    int arr[2][2] = {{1, 2}, {3, 4}};
    int *p = &arr[0][0];
    printf("%d", p[3]); // expects 4 to be printed
using the pointer returned by malloc for creating and reading heterogeneous monster-objects:

```c
void *p = malloc(sizeof(int) + sizeof(float));
int *pi = p;
float *pf = (float *)((char *)p + sizeof(int));
```

Printf format specifiers are also often used incorrectly, e.g. with regards to signedness:

```c
printf("%u", -1); // expected to print UINT_MAX
```

On the other hand there are some JITs that seem to break every rule and trigger every UB ever, but still get away with it and run fine in practice. (main2:7)
- Oh yes I do! (main2:10)
- I was quite surprised recently that clang uses undefined behavior optimizations when shifting int i = 1; until i == 0. This broke some rather odd code in Tc pdump. (main2:16)
- Signed integer overflow trips up a lot of people. (main2:18)
- packed structures, inline assembler, threads(mutexes, semaphores etc), unaligned variable access. Microsoft extensions (near, far, pascal). Gnu extensions _attribute(). (main2:21)
- All the GCC extensions. The only problem I commonly encounter with compiler optimisations is debug info being removed. I can't think of a time when it affected the behaviour of my code. Obviously I don't write clever enough code. (main2:22)
- I have documented cases where subtle but logically equivalent changes to const expressions changed compiler storage decisions (e.g., mutable data segment vs. immutable data segment on embedded systems). (main2:27)
- Old-school implementation of offsetof() used a trick of computing the addresses of structure elements via a null or other common pointer, then subtracting out the null/common pointer. This is a variation on your earlier question about out-of-bounds addresses. (main2:29)
- volatile semantics have changed. MSVC has command line switch to select between ISO volatile semantics, and semantics previously used by MS compiler. (main2:30)
- In Mercury we used GCC’s label pointers to good effect. (main2:35)
- Nameless unions are one that are just too useful.

Another is the variable length array at the end of a structure. That should really have been in the standard all along, but was left out originally because on the 8086 you wouldn’t know whether you needed to insert segment canonicalization or not. (main2:37)
- It is common to disable strict aliasing. Much code violates the strict aliasing requirements of the standard. (main2:46)
- Everything about integer representation. Everyone I know assumes 2’s-complement representation of integers.

I’ve seen code that assumes IEEE 754 representation of floating-point numbers, which I assume ISO C doesn’t require.

Lots of code assumes that function calls act like compiler barriers, meaning that reordering does not happen across them. I guess this is safe by the C11 memory model, but I haven’t verified it.

All sorts of people abuse the volatile keyword w.r.t. atomics. That is always wrong even though it often works (and IBM system software - open source, just to be clear - I’ve read uses it). (main2:48)
- I believe the standard requires use of va_list and va_args when dealing with routines using "", ..." definitions, but in practice, when all the arguments are
known to be the same type (i.e. a bunch of integers), then I’ve seen code that just uses pointer math to get the other arguments.

I know of lots of code that assumes two's-complement representation of negative integers, and lots of code that assumes overflow of signed integers will occur in a predictable way, e.g. "if (x & (1 << 31))" I think integer math overflow behavior should have been declared to be "implementation-defined" rather than "undefined". (main2:59)

- There are bunch of compiler extensions that are outside the standard. (main2:60)
- There are many, the most cool ones so far are:
  - removal of NULL checks once memcpy() is used (even if size==0):
    https://gcc.gnu.org/gcc-4.9/porting_to.html
  - removal of NULL checks if the pointer is dereferenced:
    http://blog.llvm.org/2011/05/what-every-c-programmer-should-know_14.html,
  - all sorts of aliasing (people tend to disable strict aliasing),
  - all sorts of signed %, >>; all sorts of signed overflows,
  - sequence points / happens-before / memory models,
  - everyone assumes 2-complement representation of signed integers, and many more. (main2:64)
- Integer shifts beyond bit width (main2:68)
- Two’s complement seems ubiquitous today, though the standard allows one’s complement and sign and magnitude.

The standard renders char a[3][3]; a[0][5]; undefined even though a[0][5] would be part of a. I suspect all compilers would do the right thing here.

Signed integer overflow is undefined but often abused assuming it does the expected thing. Dangerous because compilers can assume it won’t happen.

(main2:70)
- Major issues:

People make assumptions about the order of evaluation, eg. f(x(), y());

People write code that depends on the order of pointers, and is nondeterministic when allocations return pointers that compare differently from run to run.

Minor issues:

People assume we’re using Annex B (that the implementation supports IEEE floating point). Things like division by zero aren’t UB and produce valid INF/NaN values.

Borland C++Builder defaults enum size to a char. Even if you have enum constants > 256. No warnings. I consider this obsolete though.

1 << 31 and 2 << 30 and −1 << 31 should all be valid ways to produce a sign bit. The first one is pretty common, but making the other invalid seems really weird.

People assume 2’s complement in places where they don’t have it. Crypto and codecs mostly, but also found in interpreters. (For amusement, build bash with -fsanitize=signed-integer-overflow and try things like "echo ${((2**63))}" [assuming 64-bit].) (main2:74)

- All the GNU and MS extensions, with things like VLAIS and similar, are pretty widespread, breaking builds with clang (main2:75)
  - Integer overflow should be undefined value, not undefined behavior. (main2:77)
  - Plenty. (main2:86)
  - Compiler optimizations are tricky in multithreaded programming, when threads need to share some data, but it cached in register. Applying right amount of volatile is an art :) (main2:77)
  - Loops unrolling causes different results in floating point arithmetic depending on alignment, loop length, instruction set, etc. (main2:87)
  - Sometime that’s need to place constants or variables at specified memory ad
I know of a large code base that still works around a bug in uClibc that was fixed many years ago: free(NULL) would crash. I met a library that #defined printf in its header meant to be included outside, preventing others from using that function. (main2:89)

- Signed integer overflow. Relying on traps from dereferencing null and nullish (zero page) pointers. (main2:90)
- Assuming 2s complement

(mostly accidentally) assuming multiple updates between sequence points works: i += i++;

assuming that int or long int can be used to reliably store a converted pointer:

int i = (int)p; /* p is some pointer type */

assuming CHAR_BIT is 8. hahaha

QEMU documents a couple of permitted deviations from the C standard:

* you may assume that integers are 2s complement representation
* you may assume that right shift of a signed integer duplicates the sign bit (ie it is an arithmetic shift, not a logical shift)

We also tend to assume that it’s OK to shift a signed integer left such that you end up shifting a 1 into its sign bit, ie "uint32_t x = 1 << 31;" does the obvious thing (though technically it is undefined behaviour).

Representation of function pointers and object pointers: C doesn’t allow you to assume they are the same size or that they can be interconverted – this is to support systems with different data and code address spaces (common in microcontrollers). But in POSIX you usually can. (see e.g. dlsym() and the later addition of dlfunc() which improved C conformance)

- low level development (main2:97)
- Passing function pointers as void *, or casting to different function pointer types (main2:98)
- The spec and compilers treat dereference of null pointer as undefined, some code (e.g. Linux kernel) has treated this as harmless and therefore been miscompiled. Particularly worrying in environments where the system doesn’t trap actual null pointer dereferences. (main2:100)
- Encryption code written by cryptographers is typically full of invalid assumptions about signed integers. I’ve encountered driver code recently that assumes enums can represent values too large to be stored as an int. Many programs rely on computing &foo->bar being safe when foo is NULL as long as it is not dereferenced. Neither compilers nor users seem to understand the meaning of volatile. (main2:106)
- Compilers that object to left shifts of negative signed integers are very annoying. I’ve had to change lots of these into a*(1<<s) in the last few years. Thankfully the compilers mostly still emit an asl though. Used to be a common idiom. (main2:111)
- Compiler optimization seriously limits some of the things you can do in an OS, such as using for loops for timing on a system without an RTC. (main2:112)
- Most people don’t seem to understand the basic problems of signed integer overflow, or the undefined behavior when casting from double to float or integer types. People seem to assume that signed right shift will do sign extension for you even though it doesn’t have to. Also people seem to believe that bit fields in structures have certain basic behaviors but almost everything about them is undefined in the standard. Also, everyone seems to think that they understand the typing rules for enums and don’t realize that objects of enum types can be any size or signedness at the compilers whim. (main2:123)
- There are numerous unofficial (optional) features and tweaks from various embedded compilers (main2:124)
- Strict aliasing can break obvious and useful constructs, which is why I usually turn it off.

If I know my target system and don’t intend to ever target anything different, I
I'll take advantage of that knowledge in my C to do things that are normally undefined, with careful thought about compiler optimizations (this has caused me headaches in the past!).

- Wow. Not off the top of my head, no. (main2:134)
- 2’s complement arithmetic is not guaranteed by the ISO standard but generally assumed.

sizeof(char) < sizeof(int) is generally assumed.

- (a==1) should always 0 or 1 (after C89 standard).
- Altera DS5 compiler returns a instead. (main2:139)
- Assumptions about byte sizes of primitive types (especially LP64/ILP64/LLP64), endianness, everything to do with ‘const’, ‘volatile’ and ‘restrict’, the floating point environment, the signedness of character types, everything to do with array(pointer decay, nonsense initializers, mishandled signals and races, setjmp/longjmp, varargs, VLAs, alloca oh god alloca, ummm basically everything you can possibly shoot your foot with in the standard it is standard practice to shoot your foot with. (main2:156)
- Like I said before, I’ve written code (many years ago) that relied on (signed) integer arithmetic to wrap. Nowadays, I would use unsigned arithmetic instead (or perhaps use -fwrapv). I’m pretty sure I’ve also written code that violates type-based aliasing requirements.

I’ve read a lot about the efforts to create a sane memory model, to allow for safe, efficient lock-free threaded code (for example); but all my threaded code uses mutexes so the most bizarre problems with the memory models don’t really affect me. (main2:160)

- I’ve written vector (math) code that uses float x, y, z and also treats the vector as an array by using &x. I.e. This assumes structure members are contiguous.

- The C standard prescribes the type of some identifiers, such as malloc, but compilers do not enforce this uniformly. Some of those identifiers can be declared with a different type. (main2:174)

- Programmers most often assume integers cannot get close to extreme values, e.g. INT_MAX for-loop iterations, abs(INT_MIN), (a + b)/2 to compute middle value, etc. (main2:176)

- The memory order of struct members is undefined by the standard, I think. But all compilers store them in the same order that they are declared, and programmers rely on that all the time. (freebsd:0)

- Well, FreeBSD has recently had issues with signed integer overflow being treated as undefined, for instance. (freebsd:1)

- As a writer of crypto code. The only language you can write safe crypto code in now is assembly. No current or past version of C or C++ is safe to use. The compilers used today may be safe, but as we have seen in the last few years with strict aliasing and wrapv, you cannot depend upon the compiler not optimizing things away.

For example, zeroing key material when the variable is a stack (auto) variable. This has stopped working, and there are bad (and ineffective) methods for working around this issue:

http://www.daemonology.net/blog/2014-09-04-how-to-zero-a-buffer.html

The other example is for doing a constant time compare. There are implementations like:

https://cryptocoding.net/index.php/Coding_rules#Compare_secret_strings_in_constant_time

But these are explicitly unsafe in C as there is nothing per the C machine spec that requires all loops to be evaluated. It would be entirely possible for the
compiler to insert a branch statement and exit early when it knows that the value returned will be true.

Both of these cases NEED to be addressed in the C language. If they are not addressed in the C language and deployed immediately, we will see another round of bugs in the near future as compilers do more aggressive optimizations. (freebsd: 5)

- There was an interesting case involving strict aliasing and the BSD CIRCLEQ macros, see https://krbdev.mit.edu/rt/Ticket/Display.html?id=7860 which links to http://mail-index.netbsd.org/tech-kern/2013/11/20/msg016059.html, https://bugs.launchpad.net/ubuntu/+source/krb5/+bug/1347147, and https://gcc.gnu.org/bugzilla/show_bug.cgi?id=61964. This was a case where excessive (i.e., incorrect) compiler optimization caused real visible breakages. (freebsd:8)
- POSIX threads place limitations on the optimizations a C compiler can do. For example, a C compiler cannot speculate stores into conditionals. So, on UNIX-like systems, the standard is C + POSIX.

Many programmers rely on the ABI of the systems they use: for example everyone "knows" that an int is 32 bits and a pointer 32 or 64 bits. (gcc:0)
- There are many, I’m not going to remember them all off hand.

One that recently came up is that the standard forbids memcpy(p, NULL, 0), permitting the program to crash, but programs do this routinely. Compilers use this to conclude that any pointer passed to memcpy must not be NULL.

You didn’t get into overflow. Programmers routinely assume that they can test "a + b < a", where a and b are signed types, to see if a + b overflows, but of course compilers routinely assume that comparison is always false.

You only mentioned aliasing a bit. Programmers routinely assume that they can convert from int* to float* and read the bytes. This fails in many ways. (gcc:1)
- Whether >> is sign extending (usually assumed to be so for signed values).
- Assuming the layout of structs (and classes in C++)
  (google:2)
  - much older code assumes that shifting a negative integer to the right using ">>" performs an arithmetic (not bitwise) shift.

Things like random number generators often assume integer overflow is harmless and exception-free, and don’t bother making sure the overflow takes place using unsigned values. (google:7)
- I’ve seen optimization thwart integer overflow detection, but that particular problem has never applied to me. (google:21)
- I have seen programs use LoadLibrary on self to detect the presence of a feature at runtime. It does need tricks to prevent the compiler from optimizing away the code though. (google:23)
- I’ve had problems where machine generated C code would not compile. I had to break it into 1000 line blocks.
  (google:27)
- Identifier lengths —— everyone assumes that they’re unlimited, despite the standard putting maximum lengths on them. (google:36)
- Arithmetic on void pointers is interpreted as arithmetic on char pointers in gcc last i checked. (google:39)
- I used to think signed integer arithmetic would overflow and wrap around, the same way unsigned integer arithmetic does. I think modern compilers warn about this, but I don’t think that was always the case (I don’t remember getting warnings for this). (google:40)
- sizeof(void) == 1. This is a very sane and widely used fact and the standard should just adopt it already. When I do pointer arithmetic on unknown/opaque data, I obviously want to count it in bytes, and having to cast it to a type that doesn’t really have anything to do with the data itself hurts readability for no reason.

sizeof() for flexible array members should obviously be 0. They’re often used in data structures with a header and a variable-length body, and sizeof() for the whole structure is an easy and obvious way to refer to the header size.
Several GCC extensions (statement expressions, typeof, (extended) inline assembly) and attributes (aligned, packed, weak, section, used, unused) are so useful and ubiquitous that they (or equivalent functionality) should just be adopted into the standard already. (google:43)

- There are lots more differences. Don’t have time to write a treatise about it. Here’s one example:

```c
char *p = 0;
char *q = p + 0;
```

Lots of real C code assumes that this sort of thing works and that q is a null pointer, even though the C standard doesn’t guarantee this. (libc:1)
- int *p;
  *(volatile int*)p; // such access is volatile
eg ACCESS_ONCE macro in linux (the committee thinks that the normative text does not support this, but the c99 rationale has contradictory remarks, in practice it is relied on).

signed int representation: three variant is allowed but most code relies on two’s complement representation when doing bitwise operations (only historical computers do this differently).

1<<31; is common, eg in linux (a recent iso c dr asked for it to be valid after c++14 made it valid)

pointer bit representation is the same as int bit representation eg. aligning a pointer by (char*)((uintptr_t)p & −8U)

some types are assumed to be the same or have the same range (size_t and uintptr_t, or ptrdiff_t and intptr_t)

'A' == 65.

uint32_t does not promote to signed int (only true if int is 32bit or smaller).

accessing globals from signal handlers. (standard is very strict, but when the code makes sure the signal handler does not run asynchronously with a conflicting access (using signal masks) or that the signal handler never returns then it is relied on).

passing NULL as the last argument to a variadic function with "null-pointer-sentinel" like execl (in C NULL can be plain 0, posix requires (void*)0, the former can cause problems in practice on LP64 abis).

converting void* to function pointers (posix dlsym api relies on it)

iso c requires that rounding mode can be changed when #pragma STDC FENV_ACCESS ON, but compilers misoptimize such code.
(-frounding-math helps sometimes on gcc).

iso c99 has f(int p[static 1]) to mean the p argument must be non-null, but compilers don’t seem to produce diagnostic for it (instead implementation specific attributes are used for this).
(libc:3)
- Do you really have a few years to note down all the differences? (libc:5)
- Quiet two’s complement wraparound behavior of signed integers is extremely prevalent assumption, but is undefined behavior. In my opinion that is the single most ignored piece of ISO C/C++ standards. (llvm:1)
- I am not a good person to ask about the standard. I care about what works. (llvm:3)
- My only complaint about "compiler optimizations limiting the behavior I can rely upon" is in the GCC assembler for the MIPS architecture; it makes a lot of assumptions as a "high level assembler" that makes writing l1l machine code pretty difficult. But that isn’t a C compiler. ;) (llvm:7)
- A fun difference between bit-fields and explicit bit-masking and shifting. It is allowed to write to a bit-field even if its neighbor is uninitialized:

long and int are technically incompatible even if they have the same representation. This means that even if they have the same representation, strict aliasing rules apply to them, and one should be careful with strict aliasing constraints. Also, under these conditions, printf("%-ld", 1) is undefined.

printf("%-d", 1u) is also undefined.

memcmp(0, 0, 0) is forbidden, although programmers who understand the notion of zero can sometimes use it or equivalent versions in good faith. memcmp(&a+1, &b+1, 0) is also forbidden (convincing argument at http://stackoverflow.com/a/25390675/139746)

(regehr-blog:0)
- Almost all programmers expect silent two's-complement wraparound on signed integer overflow. (regehr-blog:2)
- I've heard that "volatile" is implemented poorly by all compilers and cannot be relied on. (regehr-blog:7)
- I know of some embedded code that relies on a particular behavior of signed integer overflow, though the ISO standard says it is undefined. (regehr-blog:8)
- The implementation of the offsetof macro is usually implemented through UB (null pointer dereferences, specifically). (regehr-blog:9)
- Compiler-specific "packed" structures are frequently used as if they define a stable memory layout.

Explicit dereference of NULL pointer is still used to (try to) coerce a crash (SIGSEGV).

Type punning is frequently used to try to convert, for example, a float to its bit representation in a uint32_t (as in: uint32_t x = *(uint32_t *)&float_value). This is often associated with code that is trying to explicitly handle endianness.

The POSIX dlsym() function returns a void * instead of a pointer-to-function, and the Linux man page suggests a workaround (attributed to POSIX.1-2003) of:

    double (*cosine)(double);
    *(void **) (&cosine) = dlsym(handle, "cos");

Structures are memcpy'ed from separate processes (and sometimes machines) for IPC, often with the assumption that only endianness and maybe alignment can change.

Plain char * is often used to read and write the "object representations". As I understand it, only unsigned char * is technically allowed here.

CHAR_BIT is almost always assumed to be 8 by anyone not working on DSP code.

Signed integer arithmetic is frequently treated as if overflow will produce the value that truncated 2's-complement arithmetic would produce (e.g., INT_MAX + 1 == INT_MIN). (regehr-blog:12)
- We use unsigned integers to hold scaled angle values (0 to 360 degrees), relying on unsigned integer underflow and overflow to wraparound so that we don't have to renormalize computations with angle values. (I suspect this relies on defined behavior.)
    (regehr-blog:14)
- concurrency (regehr-blog:16)
- Arithmetic on void pointers -- GNU C defines this as a language extension, a
nd I (and other codebases) use it somewhat often.

Casting between function and non-function pointer types (e.g. to assign the result of a dlsym(3) lookup to a function pointer) is also something I’d expect to work. (regehr-blog:17)

- I quite often see code where signed arithmetic is checked for overflow after the fact. This despite the fact that this is undefined behaviour and there is no reason for the result to be anything in particular (hence you can’t detect a problem after the fact).

(regehr-blog:20)
- Changes to re-ordering rules, particularly re-ordering around volatile accesses, is always a worry/fear. I don’t care what you do, please define something, so that there is a fixed definition and a fixed way of getting that behaviour. (regehr-blog:25)

- Pretty much all modern machines use 2’s complement integer types, and some IEEE-754-flavored floating-point formats. This is true even of tiny embedded chips, DSPs, etc. More standardization around things like shifting bits into the sign bit of a signed integer, would be helpful. (regehr-blog:26)

- Apart from maybe a few Pascal transliterations and textbook programs, every serious C program contains undefined behaviour, including the C compilers maintained by people who think that every undefined behaviour justifies arbitrary results. A well-known example is signed integer overflow, but I guess there are many others that we will only notice when compilers get "smart" enough to miscompile code using them. (regehr-blog:28)

- We’ve had to clean up lots of old code with issues from checking for integer overflow after the fact, where the compiler could assume it couldn't have overflowed, because you already did the operation successfully. (x11:0)
- asm! asm! asm! :) (x11:3)

- Signed bit shift and integer overflow are the biggest ones; but gcc has caused enough problems with these already that they may not matter any more.

Also, concurrency: I expect that there’s a good bit of divergence between what’s been added to C11 (which I’m not that familiar with yet) and what real threaded/multiprocessor C does. There are lots and lots of ways to get screwed by the compiler in threaded code, and not much in the way of helpful program analysis either.

Similarly, there’s a wide divergence between what C (and even POSIX) allows in signal handlers and what’s actually *in* signal handler code in deployed software. (x11:5)
- memzeroing sensitive data in in memory (x11:6)

- One of the biggest ones is bounds checking on pointer arithmetic. When you want to check to make sure you don’t run off the end of an array, the absolute most natural thing to do is something like:

```c
for(p=array; p-array < array_len; p++) { /* do something with p */ }
```

But compiler writers want to assume that all pointers are valid all the time; so since they "know" for a fact that p is a valid pointer, then of course p-array *must* be less than array_len -- no need to check it; so they just get rid of the comparison altogether.

It would be one thing if C actually did its own bounds-checking automatically. But for a language to not do any bounds-checking on its own, and then make it difficult to do "correct" bounds-checking programmatically (and very easy to think you’re doing bounds-checking and actually not be), is just insane. There have been dozens of security vulnerabilities made because of this assumption. I can’t really regard that as anything other than a bug in the compiler and/or the language specification. (xen:2)

Any other comments

Other comments:
- I don’t believe your survey is significant. People do know (and probably do expect) that the C standard is not enough, and that for practical purposes their
  C code is supposed to work on some common (not all) 32 or 64 bits processors – i.e. x86, PowerPC, ARM, Sparc – (or, if they code with great care) on some other (e.g. 16 bits) processor. C matters practically for common processors, not as an ISO standard! (main2.2:4)
- I think the standards committee should standardise how C is used in practice, which is as a fairly-low-level language, rather than trying to compete with higher-level languages.

I think compiler writers need to optimize real C programs rather than trying to find spec loopholes to exploit. Optimizing out ‘undefined’ code – e.g. by assuming that pointers from different objects can never compare equal – isn’t actually what the programmer wants the compiler to do.

We already have things like restrict that programmers can use to provide optimization hints.

If you want more restricted pointers (e.g. to allow objects to be moved by a garbage collector, which is what a lot of the questions appear to be hinting at) then I would make a new kind of type to represent the new pointers and leave normal pointers alone.

- If the standard wasn’t so insanely portable, it’d be easier to follow and less confusing.

We have modern and mostly sane architectures, but the standard still won’t accept what is de facto defined.

Thanks for the interesting questions, it was fun to think about. Looking forward to the results. (main2:7)
- A lot of the answer happen to work most of the time, even if they’re not standard. (main2:10)
- As an author on the "beyond the pdp-11" ASPLOS paper I’m obviously an outlier in this survey. :) (main2:16)
- The answers above are "in practice", not necessarily "according to the standard", which I believe is your intention. (main2:27)
- These are all evil. (main2:29)
- Working at this level of detail, I’d rather be using C++, had it succeeded. (main2:35)
- Even in the days when the 8086 was current, people ignored the bits of the C standard that were there only for the 8086 (unless you were coding for the 8086 in which case the rules were useful to you and you followed them). We should certainly not pay any attention to them today. (main2:37)
- In all cases above, I would expect the behaviour to work in practice on normal modern systems (I generally program ARM-based devices), even if not guaranteed and I might well rely on it. (main2:44)
- The C standards are fundamentally broken in many ways, see https://davmac.wordpress.com/c99-errata/. It’s hard to blame compiler vendors, and C programmers, for relying on particular interpretations of or even disregarding parts of the standard when the document itself makes little sense in certain areas. (main2:46)
- Overzealous optimizers have become a real headache when doing low-level programming. You can no longer rely on a compiler translating your code in a predictable way, and the language is not expressive enough to explain to the compiler what’s going on in an environment that’s not the comfortable sandbox of userspace to allow it to optimize safely. Few instruments are available to tame the optimizer, and they are often very blunt instruments (e.g. "volatile") with severe performance impact.

Although I understand baremetal programming is only a small portion of the C/C++ programmer audience, in practice these are the only common languages (apart from assembly) at all suitable for such tasks, and their needs seem to be overlooked.
- This quiz would be a lot better if you gave code snippets rather than vague
The vast majority of things you ask about are probably a bad idea that people do depend on. (main2:60)

I’m happy to say I haven’t poked around most of the behavior in here. I’m curious about what is permitted with respect to intptr_t. I have dabbled there before. (main2:61)

"work in normal compilers" is very open to subjective interpretation. The follow up question may bias the survey respondent to change their initial answer. (main2:70)

Fun survey.

Curious about all of the padding-related questions. Are there really people trying to do something useful with struct padding? (main2:72)

Typo, in [10/15] "intial members" should be "initial members". (main2:74)

This questionnaire is awesome. (main2:86)

There’s definitely a conflict between the part of the C community that writes high-level C code (e.g., high performance server code), and uses the "type-safe" subset of C for it, and the community that wants a portable assembler that is easy to work with (i.e., fancy language features), and that has predictable semantics. There’s also the compiler (optimizer, benchmark tuning) community, that wants to take advantage of every affordance in the language to increase the benchmark score. Also, a program analysis community that tries to leverage the same for a different purpose.

At least the way I see it, the first two goals (type safe code vs. high level assembler) are in a direct conflict. The confusion arises from semantics added for one side affecting code from the other side (aggressive optimizations causing miscompiles in low-level code, and high-level code not getting optimizations it could because of supported low-level patterns). One way to solve it would be to introduce language modes with explicit semantics for those two usecases, and an ability to switch locally. Is that the right choice for C? I don’t know. But in abstract it seems to allow us to clean up the mess we have now.

I dislike the recent tendency by compiler developers towards ‘adversarial optimisation’ which actively breaks code that is technically not standards-conformant but is playing by the ‘spirit’ of the language. I’ve spent a fair amount of time with clang’s undefined-behaviour sanitizer tools trying to locate various kinds of UB not because it’s actually causing bugs but because I feel I can no longer trust the compiler to compile a left shift operation as a simple left shift. This kind of thing might give a 0.5% improvement in specint but it’s not actually very useful for C’s real-world users...

I’m not a fan of ‘adversarial’ optimisation that assumes that programmers never write code that has undefined behaviour and then reasons from there. The result is that trivial bugs that would be easy to understand and fix get turned into code that has is spectacularly unintuitive. I’d rather take the performance hit from a less clever optimiser. (main2:99)

I don’t know anything about c (main2:105)

My answers are based on my best guess without looking up any of the answers. I guess that was what you were after? ... If I was going to write code that relied on any of these corner cases I would definitely take the time to look it up and see if it is a) legal according to the standard and b) works reliably across the target compilers for our software. (main2:108)

I’d love to see C become more well specified while still being efficient. (main2:123)

The most generic model of the computer that C encourages is that of operations on an address space represented as an array of char. If someone can figure a reason to leverage this model, they will. (main2:138)

Please take all of my opinions as coming from a noob programmer. I’ve only been coding in C for a couple years (no proper schooling or book learning during that time). (main2:141)

Sorry for being cranky, I don’t want to be rude but the questions are phrased a bit unclearly; the general point I am trying to get across is that there’s enough code that assumes uintptr_t is bit-identical to void* and bit-identical to an...
y other pointer type -- that these are all just numbers in a flat address model
-- that deviating from it in any case not forced-upon you by the target machine
is asking for obsolescence as a toolchain vendor. (main2:156)

- I assume you wanted my current beliefs (without trying to look up "correct"
  answers). Like I said several times above, in cases where I was less sure I’d t
  ry to look up the safe way to do things. (main2:160)

- Would be nice if you showed some code as illustration. (main2:176)

- C started off as a "high-level assembler" -- and that created much of its pop
  ularity. It appears that using it now requires you to be a language lawyer -- a
  minor infraction now gives the compiler license to silently turn your code to mu
  sh. If I’m writing code that is only going to run on a 32-bit 2’s complement CP
  U where arithmetic operations wrap modulo 2^32, I shouldn’t need to code on the
  assumption that my code might run on a 31-bit 1’s complement machine that genera
  tes exceptions on overflow -- and fight a compiler that has decided that if it ca
  n show my code won’t work on the latter, it’s free to do anything it wants.

Whilst dead code analysis is, in general, a good thing, there needs to be a way
to unambiguously tell the compiler "I want you to do this here, even if you don’t
think it’s necessary". The most obvious cases are:
- Overwriting buffers to ensure data is destroyed (eg key material)
- Writing and reading I/O devices where "memory" accesses have side effects.
- Implementing a delay of some sort (typically associated with the above of belo
w points)
- Writing benchmarks: They are hard enough to get right normally but when the c
  ompiler is allowed to decide that it can elide all the code because it has no ob
  vious side-effects, it becomes nearly impossible. (freebsd:6)
  - It’s hard to write code which reads chunks of data from disk and interprets
    them as different C structures (i.e., as in a database engine) while retaining s
    trict aliasing correctness. (freebsd:8)
- I don’t like the way answers are formulated.
The 5 basic answers (yes (for what), ..., don’t know) are all suggesting that th
ese constructs are bad. This is a really bad way to do a survey. (google:13)
- More wild behaviour: A hashtable implementation that used void* values <16 a
  s special. And of course Windows ShellExecute (https://msdn.microsoft.com/en-us/
  library/windows/desktop/bb762153%28v=vs.85%29.aspx) has this gem:

Return value
Type: HINSTANCE
If the function succeeds, it returns a value greater than 32. If the function fa
ils, it returns an error value that indicates the cause of the failure. The retu
rn value is cast as an HINSTANCE for backward compatibility with 16-bit Windows
applications. It is not a true HINSTANCE, however. It can be cast only to an int
and compared to either 32 or the following error codes below.
(google:14)
- A lot of these tricks are to be avoided if possible, even if they are legal.
  They make code tricky.
(google:27)
- You should have a question of "how well do you know C", because otherwise di
  fferent levels of experience and people like me who haven’t written code in it i
  n any sufficiently complicated context may muddle your data.

(google:31)
- The main issue is that less and less of such things works. For example relyi
  ng on integer overflows worked even few years ago, but now compilers can optimiz
  e things away. (google:46)
- Survey is pretty long and detailed; not sure its results will be that reliab
  le, I’m afraid. (libc:1)
- I think the options for "Do you know of real code that relies on it?" and "W
  ill that work in normal C compilers?" could have some explanation or examples.
  (libc:3)
- I consider myself an expert in C, but I wasn’t real confident about my answe
  rs to any of these questions. Maybe that means the code base I work in isn’t too
crazy, and I should be happy. (libc:7)
- I love c because I can do anything with it. (llvm:3)
- Should it be allowed to compare memory locations containing pointers with me
  mcmp?
If you answered â, should strlen be allowed on memory locations containing pointers? (regehr-blog:0)
- Hey Peter, long time no see :) (regehr-blog:1)
- Most of these questions sound like someone’s trying to write a program to analyze memory allocation strategies for the purpose of exploiting uninitialized reads: grab the mem first, write it, free it, watch it get alloc’d and read by some other function. Should I be worried? (regehr-blog:7)
- don’t optimize out null pointer tests unless you can guarantee that a potentially faulting load or store that uses the pointer will be executed before the test...

(regehr-blog:11)
- My answers may be a bit skewed by the fact that there are certain things I’ve learned to avoid doing based on knowledge of the fact that they’re technically (by the letter of the standard) undefined, even if the compiler I’m working with would in fact tolerate them and produce "sane" behavior. (regehr-blog:17)
- Interesting set of questions. As much as optimizations might affect the behavior on some of the corner cases, I’m also just as sure as there are cases where a compiler might disable optimizations to keep the corner cases valid. (regehr-blog:29)
- Very interesting survey. I’ll be excited to read the results (and probably learn why I was wrong!) (x11:2)
- no question about using compiler extensions.

Like 0b0 on gcc (x11:3)
- There’s a general pattern in most of these issues, or at least most of the issues that get coders angry and compiler writers defensive, which is that the compiler writers cite only the C standard and what’s undefined there, but coders expect compilers to continue the historic practice of honoring things that are defined by the processor architecture and reflecting these properties into C code. After all, C contains wiggle room precisely so that these properties can be engaged rather than (expensively) suppressed.

So for example people expect signed shift (particularly signed right shift), signed integer overflow, sign extension, integer to pointer conversions, pointer representations and comparisons, valid ranges of memory, endianness, and so forth to reflect the processor architecture that they’re using. They are mostly aware that not all processors are the same; but they generally don’t care much about obscure machines from decades ago. And so they get mad when the compiler fails to honor properties of this form that all current/interesting machines have in common. But this point is rarely articulated in these terms, so when the compiler writers come back and talk about undefined behavior they just grumble or get mad rather than engage the compiler writers productively.

At this stage I would be surprised to learn that anyone important working on gcc or clang really understood this point. Perhaps that’s a bit cynical. (x11:5)