P2188R1: Pointers are sometimes just bags of bits

Anthony Williams

Woven Planet
`https://www.woven-planet.global`

February 2023

woven planet

Overview

This is about 3 things:

- Programmer expectations
- Language self-consistency
- Safety

Fundamental Assumption

same bits $\Rightarrow$ same value

for objects of the same scalar type.

Source

This comes from [basic.types.general] p2, p3, and especially p4

Source

*For any object (other than a potentially-overlapping subobject) of trivially copyable type `T`, whether or not the object holds a valid value of type `T`, the underlying bytes making up the object can be copied into an array of `char`, `unsigned char`, or `std::byte`. **If the content of that array is copied back into the object, the object shall subsequently hold its original value.** (C++23 §6.8.1 [basic.types.general]/2)*

woven planet

Source

*For two distinct objects `obj1` and `obj2` of trivially copyable type `T`, where neither `obj1` nor `obj2` is a potentially-overlapping subobject, **if the underlying bytes making up `obj1` are copied into `obj2`, `obj2` shall subsequently hold the same value as `obj1`**. (C++23 §6.8.1 [basic.types.general]/3)*

woven planet

Source

… For trivially copyable types, ***the value representation is a set of bits in the object representation that determines a value****, which is one discrete element of an implementation-defined set of values. (C++23 §6.8.1 [basic.types.general]/4)*

Source

P2434 makes this clearer by talking about **bit values**.

> *Each trivially copyable type T has an implementation-defined set of discrete values. A bit value is a member of an implementation-defined disjoint partition of the set of values; for scalar types other than object pointer types, each contains no more than one value. The value representation of an object of type T determines a bit value for that object. When an object acquires a bit value, its value becomes an unspecified member of that bit value that would result in the program having defined behavior, if any. (P2434 wording for [basic.types.trivial])*

Non-assumption

This does not necessarily work the other way round

same value $\not\Rightarrow$ **same bits**

e.g. different floating point representations of the same number.

Second non-assumption

If a and b have different types,

same bits $\not\Rightarrow$ **same value**

e.g. signed vs unsigned integers, or integer vs floating point.

woven planet

Big Proviso

This only matters if the programmer "knows" that a and b have the same bits.

How do we know?

How can we “know” that a and b have the same bits?

How do we know?

How can we “know” that `a` and `b` have the same bits?

- Comparison of bits via `memcmp` or similar

How do we know?

How can we “know” that `a` and `b` have the same bits?

- Comparison of bits via `memcmp` or similar
- Direct assignment to bits via `memcpy` or similar

How do we know?

How can we “know” that `a` and `b` have the same bits?

- Comparison of bits via `memcmp` or similar
- Direct assignment to bits via `memcpy` or similar
- Comparison via an atomic operation, such as `compare_exchange`

How do we know?

How can we “know” that `a` and `b` have the same bits?

- Comparison of bits via `memcmp` or similar
- Direct assignment to bits via `memcpy` or similar
- Comparison via an atomic operation, such as `compare_exchange`
- Copying the bits to an intermediate storage (such as a suitably sized array of `byte` or a suitable integer) and comparing that storage

How do we know?

How can we “know” that `a` and `b` have the same bits?

- Comparison of bits via `memcmp` or similar
- Direct assignment to bits via `memcpy` or similar
- Comparison via an atomic operation, such as `compare_exchange`
- Copying the bits to an intermediate storage (such as a suitably sized array of `byte` or a suitable integer) and comparing that storage
- Copying the bits from `a` to `b` via intermediate storage

Consequences

same bits $\Rightarrow$ same value

means:

- **same bits** $\Rightarrow$ `a == b`
- **different values** $\Rightarrow$ **different bits**
- `a != b` $\Rightarrow$ **different bits** (or something like NaN)
- **same bits** $\Rightarrow$ **a and b are interchangeable**

Second Assumption

Numbers have no history

This follows from assumption 1, but is important enough to spell out.

Second Assumption Consequences

After a sequence of operations that yields the same integral values — **including via I/O** — integers have the same properties as if left unchanged.

Third Assumption

Comparison results for scalar types are consistent over time.

Third Assumption Consequences

- If I compare two initialized variables `a` and `b` then `a == b` should yield the same result at different points in the code, if neither `a` nor `b` has been modified in between.
- The same applies to `a != b`
- This applies to copies, and comparisons of copies in other functions.
- If the initial comparison was, or arose from, undefined behaviour then all bets are off anyway.

Fourth Assumption

The bits of an object don't change unless the object is modified.

Modifications to an object include modification via pointer or reference, or by modifying the bits of its object representation directly.

Fourth Assumption Consequences

- Operations on other objects cannot modify an object
- Copying the object representation of `a` twice yields the same sequence of `byte` objects unless `a` was modified in between

Pointers

Pointers are **scalar types** and **trivially copyable types** (basic.types.general p9).

Pointers

Pointers are **scalar types** and **trivially copyable types** (basic.types.general p9).

Therefore all the previous points apply to them.

Pointers

Pointers are **scalar types** and **trivially copyable types** (basic.types.general p9).

Therefore all the previous points apply to them.

Any case where this does not hold breaks the fundamental assumptions, and makes C++ internally inconsistent at the lowest level.

Consequences for Pointers

- If I compare the bits of two pointers, **and they are equal**, then I can use either in the place of the other from then on (assumption 1).

Consequences for Pointers

- If I compare the bits of two pointers, **and they are equal**, then I can use either in the place of the other from then on (assumption 1).
- If I compare the bits of two pointers, **and they are equal**, then the pointers had better be equal too (assumption 1).

Consequences for Pointers

- If I compare the bits of two pointers, **and they are equal**, then I can use either in the place of the other from then on (assumption 1).
- If I compare the bits of two pointers, **and they are equal**, then the pointers had better be equal too (assumption 1).
- If I cast a pointer to an integer, and that integer compares equal to an integer I obtained from elsewhere, casting that second integer to a pointer yields a pointer I can use in place of the original (assumptions 1 and 2).

Consequences for Pointers

- If I compare the bits of two pointers, **and they are equal**, then I can use either in the place of the other from then on (assumption 1).
- If I compare the bits of two pointers, **and they are equal**, then the pointers had better be equal too (assumption 1).
- If I cast a pointer to an integer, and that integer compares equal to an integer I obtained from elsewhere, casting that second integer to a pointer yields a pointer I can use in place of the original (assumptions 1 and 2).
- Deleting an object does not change the bits of a pointer to that object (assumption 4).

woven planet

Consequences for Pointers

- If I compare the bits of two pointers, **and they are equal**, then I can use either in the place of the other from then on (assumption 1).
- If I compare the bits of two pointers, **and they are equal**, then the pointers had better be equal too (assumption 1).
- If I cast a pointer to an integer, and that integer compares equal to an integer I obtained from elsewhere, casting that second integer to a pointer yields a pointer I can use in place of the original (assumptions 1 and 2).
- Deleting an object does not change the bits of a pointer to that object (assumption 4).
- If I compare two pointers then the result of that comparison is unchanged if I repeat it, **even if one of the pointed-to objects is deleted** (assumptions 1, 3 and 4)

Interaction with Provenance

- Alias analysis relies on the **provenance** of a pointer to determine whether or not the object referenced by a pointer can **alias** another.

Interaction with Provenance

- Alias analysis relies on the **provenance** of a pointer to determine whether or not the object referenced by a pointer can **alias** another.
- If the analysis cannot prove that two pointers cannot alias, then it must assume that they may alias.

Interaction with Provenance

- Alias analysis relies on the **provenance** of a pointer to determine whether or not the object referenced by a pointer can **alias** another.
- If the analysis cannot prove that two pointers cannot alias, then it must assume that they may alias.
- If two pointers compare equal, or their bits compare equal, they must both take on the union of their two provenances.

Interaction with Provenance

- Alias analysis relies on the **provenance** of a pointer to determine whether or not the object referenced by a pointer can **alias** another.
- If the analysis cannot prove that two pointers cannot alias, then it must assume that they may alias.
- If two pointers compare equal, or their bits compare equal, they must both take on the union of their two provenances.
- If the presence or absence of such a comparison is invisible to the compiler, then it must assume that there was such a check.

Interaction with Provenance

- Alias analysis relies on the **provenance** of a pointer to determine whether or not the object referenced by a pointer can **alias** another.
- If the analysis cannot prove that two pointers cannot alias, then it must assume that they may alias.
- If two pointers compare equal, or their bits compare equal, they must both take on the union of their two provenances.
- If the presence or absence of such a comparison is invisible to the compiler, then it must assume that there was such a check.
- This may require assigning “wildcard” provenance.

woven planet

Examples

Example: Avoiding spooky action at a distance

```
struct X {
  X* next;
  int value;
};
std::atomic<X*> top{nullptr};

void push(int v) {
  X* nv = new X{top.load(), v}; // A
  while(!top.compare_exchange_strong(nv->next, nv)) // B
  {}
}

int pop() {
  X* p = top.load(); // C
  while(p &&
    !top.compare_exchange_strong(p, p->next)) // D
  {}
  if(!p) throw std::runtime_error("Empty");
  int retval = p->value;
  delete p; // E
  return retval
}
```

woven planet

Example: Avoiding spooky action at a distance

```
struct X {
  X* next;
  int value;
};
std::atomic<X*> top{nullptr};

void push(int v) {
  X* nv = new X{top.load(), v}; // A
  while(!top.compare_exchange_strong(nv->next, nv)) // B
  {}
}

int pop() {
  X* p = top.load(); // C
  while(p &&
    !top.compare_exchange_strong(p, p->next)) // D
  {}
  if(!p) throw std::runtime_error("Empty");
  int retval = p->value;
  delete p; // E
  return retval
}
```

If `push` and `pop` are called from different threads, the execution may be A -> C -> D -> E -> B

`nv->next` on line B is thus an **invalid pointer value, even though it is not modified**.

`nv->next` must maintain its behaviour from the point of view of comparisons. Either it succeeds (which means that it has become a pointer to a newly allocated `X` with the same address), or it fails, so we can't rely on it being a pointer to a valid `X` object.

Example: Avoiding spooky action at a distance

In real code, the compare-exchange might succeed because a third thread called `pop` and the new `X` has the same address as the old one. The execution sequence is thus A -> C -> D -> E -> A' -> B' -> B

In such a scenario, subsequent calls to `pop` must correctly yield the new pointer to the new object allocated at A', and not trigger UB when evaluating `p->value` due to the pointer referring to the deleted `X` object.

This behaviour is expected from assumptions 1, 3 and 4.

Example: Avoiding spooky action at a distance

If I replace `std::atomic` with `my::atomic` that holds a plain pointer and an internal mutex, this should still work.

$\Rightarrow$ creating a special-case for `std::atomic` is insufficient.

Example: Past the end pointers

```c
int x = 1;
int y = 2;

int main() {
    int *p1 = &x + 1; // one past the end
    int *p2 = &y; // separate object
    if(memcmp(p1, p2, sizeof(p1))) { // check
        printf("Different address\n");
        return 0;
    }

    assert(p1 == p2);
    assert(*p1 == 2);
    assert(*p2 == 2);
    *p1 = 42;
    assert(*p1 == 42);
    assert(*p2 == 42);
}
```

Example: Past the end pointers

```
int x = 1;
int y = 2;

int main() {
    int *p1 = &x + 1; // one past the end
    int *p2 = &y; // separate object
    if(memcmp(p1, p2, sizeof(p1))) { // check
        printf("Different address\n");
        return 0;
    }

    assert(p1 == p2);
    assert(*p1 == 2);
    assert(*p2 == 2);
    *p1 = 42;
    assert(*p1 == 42);
    assert(*p2 == 42);
}
```

After the check, we “know” whether `p1` and `p2` have the same bits. If they don’t, we exit. If they do, then we can rely on **same bits ⇒ same value**, so the assertions must not fire.

After the check, `p1` must be treated as having the same **provenance** as `p2`. They are both **simultaneously** a “past-the-end” pointer for `x`, **and** a pointer to `y`.

If we did not check, then `*p1` would be dereferencing a past-the-end pointer (only), which is UB.

This is only safe because we check.

Examples from the paper

Example 8: `delete` and `new` the object without replacing pointer

```
struct X {
    int i;
};

int main() {
    X *x= new X{42};
    X *y= nullptr;
    unsigned char buffer[sizeof(x)];
    memcpy(buffer, &x, sizeof(x));

    delete x; // deallocate
    y= new X{99}; // allocate

    unsigned char buffer2[sizeof(x)];
    memcpy(buffer2, &y, sizeof(x));
    if(memcmp(buffer, buffer2, sizeof(x))) { // check
        printf("Different address\n");
        return 0;
    }

    assert(x == y);
    assert(y->i == 99);
    assert(x->i == 99);
}
```

woven planet

Example 8: `delete` and `new` the object without replacing pointer

```
struct X {
    int i;
};

int main() {
    X *x= new X{42};
    X *y= nullptr;
    unsigned char buffer[sizeof(x)];
    memcpy(buffer, &x, sizeof(x));

    delete x; // deallocate
    y= new X{99}; // allocate

    unsigned char buffer2[sizeof(x)];
    memcpy(buffer2, &y, sizeof(x));
    if(memcmp(buffer, buffer2, sizeof(x))) { // check
        printf("Different address\n");
        return 0;
    }

    assert(x == y);
    assert(y->i == 99);
    assert(x->i == 99);
}
```

After the check, we “know” whether `x` and `y` have the same bits. If they don’t, we exit. If they do, then we can rely on **same bits ⇒ same value**, so the assertions must not fire.

After the check, `x` must be treated as having the same **provenance** as `y`.

This may require giving it “wildcard” provenance if `buffer` is passed to an unknown function (e.g. if `memcmp` was `user_memcmp`)

Alternatively: ensure the allocator never reuses addresses, so the check always fails.

woven planet

Example 9: using `std::atomic` to hold the pointer

```
struct X {
    int i;
};

int main() {
    X *x= new X{42};
    X *y= nullptr;
    std::atomic<X *> p(x);

    delete x; // deallocate
    y= new X{99}; // allocate

    X *temp= y;
    if(!p.compare_exchange_strong(temp, y)) { // check
      printf("Different address\n");
      return 0;
    }

    assert(x == y);
    assert(y->i == 99);
    assert(x->i == 99);
}
```

woven planet

Example 9: using `std::atomic` to hold the pointer

```
struct X {
    int i;
};

int main() {
    X *x= new X{42};
    X *y= nullptr;
    std::atomic<X *> p(x);

    delete x; // deallocate
    y= new X{99}; // allocate

    X *temp= y;
    if(!p.compare_exchange_strong(temp, y)) { // check
      printf("Different address\n");
      return 0;
    }

    assert(x == y);
    assert(y->i == 99);
    assert(x->i == 99);
}
```

This is much more representative of real code. The check is done in `compare_exchange_strong`, but the consequence is the same.

If `p` is different from `temp`, then we exit.

If we don't exit, then `p` (which holds `x`) is the same as `temp` (which holds `y`), so `x` and `y` have the **same bits**, and thus the **same value**, so the asserts must not fire.

woven planet

Remaining examples from the paper

Comparisons 1: Pointers that compare equal should be interchangeable

```
void f(int* p,int* q){
  *q=99;
  bool same=false;
  if(p==q){
    *p=42;
    same=true;
    assert(*q==42);
  }

  assert(same?(*q==42):(*q==99));
}
```

Comparisons 2: Pointer equality is consistent

```
bool compare(int* const p, int* const q){
  return p==q;
}

void f(int* const p, int* const q){
  bool const same=(p==q);
  g(p,q);
  assert(same==(p==q));
  assert(same==compare(p,q));
}
```

Comparisons 3: Compilers must be able to assume no aliasing in certain circumstances

```
void f(){
  int x=42;
  g();
  assert(x==42);
}
```

Comparisons 4: Validity of pointers is contagious after comparison

```
void f(){
  int * const p=new int(42);
  delete p;
  int * const q=new int(99);

  if(p==q){
    assert(*p==99);
  }
}
```

Example 1: `memcpy` on a pointer

```
int main() {
    int *x= new int(42);
    int *y= nullptr;
    memcpy(&y,&x,sizeof(x));
    assert(x == y);
    assert(*y==42);
}
```

Example 2: `memcpy` via buffer

```
int main() {
    int *x= new int(42);
    int *y= nullptr;
    unsigned char buffer[sizeof(x)];
    memcpy(buffer, &x, sizeof(x));
    memcpy(&y, buffer, sizeof(x));
    assert(x == y);
    assert(*y == 42);
}
```

Example 3: `reinterpret_cast` to an integer

```
int main() {
    int *x= new int(42);
    int *y= nullptr;
    uintptr_t temp= reinterpret_cast<uintptr_t>(x);
    y= reinterpret_cast<int *>(temp);
    assert(x == y);
    assert(*y == 42);
}
```

Example 4: `memcpy` with modification

```
int main() {
    int *x= new int(42);
    int *y= nullptr;
    unsigned char buffer[sizeof(x)];
    memcpy(buffer, &x, sizeof(x));
    for(auto &c : buffer) {
        c^= 0x55;
    }
    for(auto &c : buffer) {
        c^= 0x55;
    }
    memcpy(&y, buffer, sizeof(x));
    assert(x == y);
    assert(*y == 42);
}
```

Example 5: `memcpy` and write to a file

```
int main() {
    int *x= new int(42);
    int *y= nullptr;
    unsigned char buffer[sizeof(x)];
    memcpy(buffer, &x, sizeof(x));

    auto file= fopen("tempfile", "wb");
    auto written= fwrite(buffer, 1, sizeof(buffer), file);
    assert(written == sizeof(buffer));
    fclose(file);

    memset(buffer, 0, sizeof(buffer));

    file= fopen("tempfile", "rb");
    auto read= fread(buffer, 1, sizeof(buffer), file);
    assert(read == sizeof(buffer));
    fclose(file);
    memcpy(&y, buffer, sizeof(x));
    assert(x == y);
    assert(*y == 42);
}
```

Example 6: destroy and recreate the object

```
struct X {
    int i;
};

int main() {
    X *x= new X{42};
    X *y= nullptr;
    unsigned char buffer[sizeof(x)];
    memcpy(buffer, &x, sizeof(x));

    x->~X();
    new(x) X{99};

    memcpy(&y, buffer, sizeof(x));
    assert(x == y);
    assert(y->;i == 99);
    assert(x->i == 99);
}
```

woven planet

Example 7: delete and new the object

```
struct X {
    int i;
};

int main() {
    X *x= new X{42};
    X *y= nullptr;
    unsigned char buffer[sizeof(x)];
    memcpy(buffer, &x, sizeof(x));

    delete x;
    y= new X{99};

    unsigned char buffer2[sizeof(x)];
    memcpy(buffer2, &y, sizeof(x));

    if(memcmp(buffer, buffer2, sizeof(x))) {
        printf("Different address\n");
        return 0;
    }

    memcpy(&x, buffer2, sizeof(x));

    assert(x == y);
    assert(y->i == 99);
    assert(x->i == 99);
}
```