Doc. No. P 3650R0
Date: 2025-03-06
Audience: EWG
Author: Bjarne Stroustrup
Reply to: bjarne@stroustrup.com

21st Century C++

Columbia University
Bjarne Stroustrup (bjarne@stroustrup.com)

Abstract

It is now 45+ years since C++ was first conceived. As planned, it evolved to meet challenges, but many developers use C++ as if it was still the previous millennium. This is suboptimal from the perspective of ease of expressing ideas, performance, reliability, and maintainability. Here, I present the key concepts on which performant, type safe, and flexible C++ software can be built: resource management, life-time management, error-handling, modularity, and generic programming. At the end, I present ways to ensure that code is contemporary, rather than relying on outdated, unsafe, and hard-to-maintain techniques: guidelines and profiles.

This paper can be read by anyone with a good knowledge of modern programming but is primarily aimed at people with a basic knowledge of C++. It is an exposition of ideals, principles, and techniques with short examples, rather than a tutorial.

1. Introduction

C++ is a language with a long history. This leads many developers, teachers, and academics to overlook decades of progress and describe C++ as if today was still the second millennium where phones had to be plugged into walls and most code was short, low-level, and slow.

If your operating system has maintained compatibility over decades, you can run C++ programs written in 1985 today on a modern computer. Stability – compatibility with earlier versions of C++ – is immensely important, especially for organizations that maintain software systems for decades. However, in essentially all cases, contemporary C++ [BS2022] can express the ideas embodied in such old-style code far simpler, with much better type-safety guarantees, and have them run faster using less memory.

This paper presents the key contemporary C++ mechanism designed to enable that. At the end (§6), it describes techniques for enforcing such modern use of C++.

Consider a simple program that writes every unique line from input to output:

```
import std;                             // make all of the standard library available
using namespace std;

int main()                              // print unique lines from input
{
        unordered_map<string,int> m;    // hash table
        for (string line; getline(cin,line); )
                if (m[line]++ == 0)
                        cout << line << '\n';
}
```

Connoisseurs will recognize this as the AWK program **(!a[$0]++)**. It uses an **unordered_map**, the C++ standard library version of a hash table, to hold unique lines and output only when a line is seen the first time.

The **for**-statement is used to limit the scope of the loop variable (**line**) to the loop.

Compared to older C++ styles, what is notable is the absence of explicit

- Allocation/deallocation
- Sizes
- Error handling
- Type conversions (casts)
- Pointers
- Unsafe subscripting
- Preprocessor use (in particular, no **#include**)

Still, this program is quite efficient compared to older styles – more efficient than most programmers could write given a reasonable amount of time. If more performance is needed it can be tuned. One important aspect of C++ is that code with a reasonable interface can be tuned to specific needs and even to use specialized hardware. This can be done without disturbing other code and often without modifying compilers.

Consider a variant of that program that collects unique lines for later use:

```
import std;                             // make all of the standard library available
using namespace std;

vector<string> collect_lines(istream& is)     // collect unique lines from input
{
        unordered_set<string> s;                        // hash table
        for (string line; getline(is,line); )
```

```
                s.insert(line);
        return vector{from_range, s};       // copy set elements into a vector
    }

    auto lines = collect_lines(cin);
```

Since I didn’t need a count, I used a set, rather than a map. I returned a **vector** rather than a **set** because **vector** is the most widely used container. I didn’t have to specify the **vector**’s element type because the compiler deduced it from the **set**’s element type.

I used the **from_range** argument to tell the compiler and a human reader that a range is used, rather than other possible ways of initializing the **vector**. I would have preferred to use the logically minimal **vector{m}** but the standards committee decided that requiring **from_range** would be a help to many.

Experienced programmers will notice that this **collect_lines()** copies the characters read. That could be a performance problem, so in §3.2, I show how to tune **collect_lines()** to avoid that.

What’s the point of these small examples? To show some contemporary C++ before going into technical details and hopefully to shake some people out of their decades-old complacent miscomprehensions.

2. C++ Ideals

My aims for C++ can be summarized as

- Direct expression of ideas
- Static type safety
- Resource safety (aka “no leaks”)
- Direct access to hardware
- Performance (aka efficiency)
- What you don't use, you don't pay for (the zero-overhead rule)
- Affordable extendibility (aka zero-overhead abstraction)
- Concurrency support (through libraries supported by intrinsics)
- Maintainability (aka comprehensible code)
- Platform independence (aka portability)
- Stability (aka compatibility)

This has not changed since the earliest days [BS1994], but C++ was meant to evolve, and contemporary C++ can deliver such properties in code much better than earlier versions of C++.

C++ code embodying these ideals is not achieved simply by applying all the latest features and only those. Some key features and techniques are old

- Classes with constructors and destructors
- Exceptions
- Templates
- **std::vector**
- …

Other key features are more recent

- Modules (§4)
- Concepts (for specifying generic interfaces; §5.1)
- Lambda expressions (for generating function objects; §5.1)
- Ranges (§5.1)
- Constexpr and consteval (for compile-time computation; §5.2)
- Concurrency support and parallel algorithms
- Coroutines (missing for decades though they were an essential part of early C++)
- **std::shared_ptr**
- …

What matters is to use the language and library features as a coherent whole in ways that suits the problem to be solved.

The value of a programming language is in the range and quality of its applications. For C++, the evidence is its amazing range of application areas: foundational software (including operating systems and databases), graphics, scientific applications, movies, games, automobiles, language implementation (and not just C++ implementation), flight controls, search engines, browsers, semiconductor design and manufacture, space probes, finance, AI, and much, much more. Given the billions of lines of C++, we cannot change the C++ language incompatibly. However, we can change the way C++ is used.

In the rest of this paper, I focus on

- §3. Resource management (including control of lifetime and error handling)
- §4. Modules (including eliminating the preprocessor)
- §5. Generic programming (including concepts)
- §6. Guidelines and enforcement (to guarantee that we write “21st Century C++”?)

Naturally, that is not all C++ offers, and much good code is written in ways not listed here. For example, I left out object-oriented programming because many developers know how to do that well in C++. Also, very high-performance code and code manipulating hardware directly require special attention and techniques. The extensive C++ concurrency support deserves at least a separate paper. However, the key to most good software is type-safe interfaces that contains sufficient information to allow optimization and run-time checking of properties that cannot be guaranteed at compile-time.

3. Resource Management

A resource is anything we must acquire and later release (give back) explicitly or implicitly. For example, memory, locks, file handles, sockets, thread handles, transactions, and shaders. To avoid resource leaks, we must avoid manual/explicit release. People – even good programmers – are notoriously bad at remembering to return what they borrowed.

The basic C++ technique for managing a resource is to root it in a handle that is guaranteed to release the resource when the handle's scope is left. For reliability, we cannot rely on explicit **delete**, **free()**, **unlock()**, etc. in application code. Such operations belong in resource handles. Consider:

```
template<typename T>
class Vector {              // vector of elements of type T
public:
        Vector(initializer_list<T>);     // constructor: acquire memory; initialize elements
        ~Vector();                       // destructor: destroy elements; release memory
        // …
private:
        T* elem;        // pointer to elements
        int sz;         // number of elements
};
```

Here, **Vector** is a resource handle. It raises the level of abstraction from the machine-near pointer plus a count of elements to a proper type with guaranteed initialization (the constructor) and clean-up (the destructor). The standard-library **vector** that this **Vector** is meant to illustrate additionally provides comparisons, assignments, more ways of initializing, resizing, support for iteration, etc. That provides the programmer with a **vector** that from a language-technical point of view (e.g., naming, scope, syntax) behaves like a built-in integer type despite being a (standard-library) resource handle and having very different semantics. We can use it like this:

```
void fct()
{
        vector<double> constants {1, 1.618, 3.14, 2.99e8};
        vector<string> designers {"Strachey", "Richards", "Ritchie"};
        // …
        vector<pair<string,jthread>> vp { {"producer",prod}, {"consumer",cons}};
}
```

Here, **constants** is initialized by four mathematical and physical values, **designers** by three hopefully well-known programming language designers, and **vp** by a producer-consumer pair. All are initialized by the appropriate constructors and released upon scope exit by the appropriate destructors. The initialization and release done by constructors and destructors is recursive. For example, the construction and destruction of **vp** is non-trivial since it involves a **vector**, a **pair**, **string**s (handles to characters), and **jthread**s (handles to operating system threads). However, it is all handled implicitly.

This use of constructor-destructor pairs (often called RAII – "Resource Acquisition Is Initialization") doesn't just guarantee release of resources, it also minimizes resource retention, thus providing a significant performance advantage compared to many other techniques, such as memory management relying on a garbage collector.

3.1. Control of lifetime

Controlling the lifetime of objects representing resources is necessary for simple and efficient resource management. C++ offers 4 points of control defined as operations on a class (here named **X**):

- ***Construction***: Invoked before first use; establish the class invariant (if any). Name: constructor, **X(**optional_arguments**)**
- ***Destruction***: Invoked after last use; release every resource (if any). Name: destructor, **~X()**
- ***Copy***: make a new object with the same value as another; **a=b** implies **a==b** (for regular types). Names: copy constructor, **X(const X&)** and copy assignment, **X::operator=(const X&)**
- ***Move***: Move resources from one object to anothert, often between scopes. Names: move constructor, **X(X&&)** and move assignment, **X::operator=(X&&)**

For example, we might elaborate our **Vector** like this:

```
template<typename T>
class Vector {          // vector of elements of type T
public:
        Vector();                         // default constructor: make an empty vector
        Vector(initializer_list<T>);      // constructor: acquire memory; initialize elements
        Vector(const Vector& a);          // copy constructor: copy a into *this
        Vector& operator=(const Vector& a);   // copy assignment: copy a into *this
        Vector(Vector&& a);                   // move constructor: move a into *this
        Vector& operator=(Vector&& a);        // move assignment: move a into *this
        ~Vector();                        // destructor: destroy elements; release memory
```

```
		// …
	};
```

Assignment operators must release any resources owned by the target object. Move operations must transfer all resources to the target and ensure that they are no longer in the source.

3.2. Eliminating redundant copies

Given this framework, let's have another look at the **collect_lines** example from §1. First, we can simplify it a bit:

```
	vector<string> collect_lines(istream& is)	// collect unique lines from input
	{
		unordered_set<string> s {from_range,istream_iterator<Line>{is}};
		return vector{from_range,s};
	}

	auto lines = collect_lines(cin);
```

The **istream_iterator<Line>{is}** allows us to treat the input from **is** as a range of **Line**s, rather than laboriously explicitly apply input operations to the stream.

Here, the **vector** is moved out of **collect_lines()** rather than copied. Since 1983 or so, compilers have known to construct the returned value (here, **vector{from_range,s};**) in the target (here, **lines**). This is referred to as "copy elision." From 2017 onwards, this has become part of the standard.

The worst-case cost of a **vector**'s move constructor is 6 word copies, three to copy the representation and three to zero-out the original representation. This is still the case if the **vector** has a million elements. For a move assignments, it's that plus the cost of destroying the target value. Moving saves us from much explicit, expensive, and error-prone explicit memory management using pointers.

By default, **string**s are read as whitespace-separated sequences of characters, but I wanted Lines read as lines. There is no **Line** type in the standard library, so I defined my own (using classical object-oriented programming):

```
	struct Line : string { };
	istream& operator>>(istream& is, Line& ln) { return getline(is, ln); }
```

I still wanted to return a **vector<string>** from **collect_lines** so I had to say so or I would have returned a **vector<Line>**.

However, the characters of the **string**s are still copied from the set into the **vector**. That could be costly. In principle, the compiler could deduce that we don't use **s** again after making the **vector** and just move the **string** elements, but today's compilers are not that smart, so we must explicitly ask for a move:

```
vector<string> collect_lines(istream& is)      // collect unique lines from input
{
        unordered_set<string> s {from_range,istream_iterator<Line>{is}};
        return vector{from_range,std::move(s)};      // move elements, rather than copy
}
```

This still leaves one redundant copy, the copy of characters from the input buffer into the set's string elements. If that's a problem, we can eliminate that also. That would involve conventional low-level techniques, simple abstractions (such as **span** (§6.2)), and compile-time evaluation (§5.2). Showing code for that is beyond the scope of this paper. Such code is messier but, as always, C++ code with well-specified interfaces is tunable. Also, please remember not to optimize without measurement showing a need to. With contemporary C++, my approach to optimizing tends to be to simplify. Code involving complex data structures and clever use of pointers tends not just to confuse programmers, but also optimizers.

3.3. Resources and errors

One of the key aims of C++ is resource safety: no resource is leaked. This implies that we must prevent resource leaks in error conditions. The basic rules are:

- Don't leak a resource
- Don't leave a resource in an invalid state

So, when an error that cannot be handled locally is detected, before exiting a function we must:

- Put every object accessed into a valid state
- Release every object that the function is responsible for
- Leave it to some function up the call chain to deal with resource-related problems

This implies that "raw pointers" cannot reliably be used as resource handles. Consider a type **Gadget** that may hold resources such as memory, locks, and file handles:

```
void f(int n, int x)
{
        Gadget g {n};
        Gadget* pg = new Gadget{n};                    // explicit new: don't!
        // …
        if (x<100) throw std::runtime_error{"Weird!"};     // leaks *pg; but not g
        if (x<200) return;                          // leaks *pg; but not g
        // …
    }
```

The explicit use of **new** to place the **Gadget** on the heap is a problem the picosecond its result is stored in a "raw pointer" rather than in a resource handle with a suitable destructor. Also, local objects are simpler and typically faster than use of explicit **new**.

For a reliable system, we need an articulated policy for handling errors. The best way to do that in general is to distinguish errors that can be handled locally by an immediate caller from errors that can be handled only far up a call chain by code with a broader view of a system's requirements for error handling:

- Use error codes and tests for failures that are common and can be handled locally.
- Use exceptions for failures that are rare ("exceptional") and cannot be handled locally:
    - To handle errors in constructors, operators, and other functions that don't have an easy way to return an error indicator (e.g., **Matrix x = y+z;**).
    - To ensure that failure to check for an error gives termination, rather than wrong results.
    - To automatically propagate errors up the call chain to a handler.
    - The alternative is expensive "error-code hell" where every caller up the call stack must remember to test.

In some important applications, unconditional immediate termination isn't an option. Then, we must remember to test every error return code and to catch every exceptions somewhere (e.g., in **main()**) and do whatever appropriate response is required.

To the surprise of many, exceptions can be cheaper and faster than consistent use of error codes even for small systems [KE2024; GCC2024].

Consider:

```
    void fct(jthread& prod, jthread& cons, string name)
    {
        ifstream in { name };
        if (!in) { /* … */ }            // possible failure expected
        // …
        vector<double> constants {1, 1.618, 3.14, 2.99e8};
        vector<string> designers {"Strachey", "Richards", "Ritchie"};
        auto dmr = "Dennis M. " + designers[2];
        // …
        pair<string,jthread&> pipeline[] { {"producer", prod}, {"consumer", cons}};
        // …
```

```
}
```

How many tests would we need for this (artificial, but not unrealistic) small example if we couldn't rely on exceptions? The example involves memory allocation, nested construction, an overloaded operator, and acquisition of a system resource.

Unfortunately, exceptions have not been universally appreciated and used everywhere they would have been most appropriate. In additions to overuse of "naked" pointers, it has been a problem that many developers insist to use a single technique for reporting all errors. That is, all reported by throwing or all reported by returning an error code. That doesn't match the needs of real-world code.

4. Modularity

The preprocessor that C++ inherited from C is essentially universally used, but it is a major obstacle to tool development, a drag on compiler performance, and a source of errors. In contemporary C++, macros used to express constants, functions, and types have been replaced with properly typed and scoped constants, compile-time evaluated functions (§5.3), and templates (§5.1). However, the preprocessor has been essential for expressing a weak form of modularity. Interfaces to libraries and other separately compiled code is represented as files containing C++ source text and **#include**d.

4.1. Header files

An **#include** directive copies the source text from such a "header file" into the current translation unit. Unfortunately, this implies that

```
#include "a.h"
#include "b.h"
```

Might have a different meaning than

```
#include "b.h"
#include "a.h"
```

This is the source of subtle bugs.

An **#include** is transitive. That is, if **a.h** contains an **#include "c.h"** the text of **c.h** also becomes part of every source file that use **#include "a.h"**. This is a source of subtle bugs. Since a header file is often **#include**d in dozens or hundreds of source files, this also implies much repeated compilation.

4.2. Modules

These problems with the use of header files to fake modularity have been known from before C++ was born, but defining an alternative and introducing it into billions of lines of code is not trivial. However, C++ now offers modules that deliver proper modularity. Importing modules is order independent, so

```
import a;
import b;
```

means the same as

```
import b;
import a;
```

The mutual independence of modules implies improved code hygiene. It makes the subtle dependency bugs impossible.

Here is a very simple example of a module definition:

```
export module map_printer;          // we are defining a module

import iostream;               // we import modules needed for the implementation
import containers;
using namespace std;

export                         // this template is the only entity exported
void print_map(const Sequence auto& m) {          // generic function; see §5
        for (const auto& [key,val] : m)           // access key and value pair from m
                cout << key << " -> " << val << '\n';
}
```

Because **import** is not transitive, users of **map_printer** do not gain access to the implementation details needed for **print_map**.

A **module** needs to be compiled once only, independently of how many times it is **import**ed. This implies very significant improvements in compile time. A user reports [DE2021]:

```
#include <libgalil/DmcDevice.h>                 // 457440 lines after preprocessing

int main() {                                    // 151268 non-blank lines
        Libgalil::DmcDevice("192.168.55.10");   // 1546 milliseconds to compile
}
```

That’s 1.5 seconds to compile almost ½ million lines of code. That’s fast! However, the compiler is doing far too much work.

```
import libgalil;                                    // 5 lines after preprocessing

int main() {                                        // 4 non-blank lines
        Libgalil::DmcDevice("192.168.55.10");       // 62 milliseconds to compile
}
```

That’s a 25 times speedup. We cannot expect that in all cases, but a 7-to-10 times advantage to **import** over **#include** is common. If you **#include**d that library in 25 source files. That would cost more than half a minute where the **import** would clock in at 1.5 seconds.

The complete standard library has been made into a module. Look at the traditional “hello world!” program [BS2021]:

```
#include <iostream>

int main()
{
        std::cout << "Hello, World!\n";
}
```

On my laptop, it compiled in 0.87 seconds. Replace the **#include<iostream>** with **import std;** and the compile time dropped to 0.08 seconds despite that it makes the complete standard library available (at least 10 times as much information).

Reorganizing significant amount of code isn’t easy or cheap, but in the case of modules, the benefits are significant in terms of code quality and massive in terms of build times.

5. Generic Programming

Generic programming is a key foundation of contemporary C++. It has been so since before “C with Classes“ was renamed “C++” but only recently (C++20) has the language support approximated the ideals [BS2022].

Generic programming, that is, programming with types and functions parameterized by types, offers

- Shorter and more readable code
- More direct expression of ideas
- Zero-overhead abstraction

- Type safety

This implies less code, easier testing, simpler maintenance, and sometimes even improved performance compared to alternatives.

Templates, the C++ language support for generic programming, are pervasive in the standard library

- Containers and algorithms
- Concurrency support: threads, locks, …
- Memory management: allocators, resource handles (e.g., **vector** and **list**), **unique_ptr**, …
- I/O
- Strings and regular expressions
- And much more

We can write code that works for all suitable argument types. For example, here is a **sort** function that accepts all types that meet the ISO C++ standard's definition of a sortable range:

```
void sort(Sortable_range auto& r);

vector<string> vs;
// … fill vs …
sort(vs);

array<int,128> ai;
// … fill ai …
sort(ai);
```

The compiler has enough information to verify that the types of **vs** and **ai** have what **Sortable_range** requires; that is, a random-access range of values of types that can be compared and swapped as needed for sorting. If the arguments are not suitable, the error is caught by the compiler at the point of use. For example:

```
list<int> lsti;
// … fill lsti …
sort(lsti);        // error: a list doesn't offer random access
```

According to the C++ standard, a **list** isn't a sortable range because it doesn't offer random access.

5.1. Templates

C++'s support for generic programming is based on three aims:

- ***Extremely general/flexible***: "it must be able to do much more than I can imagine"
- ***Zero-overhead***: Abstractions such as **vector** and **Matrix**, can compete with C arrays
- ***Well-specified interfaces***: Implying overloading and good error messages

Consider a skeleton of a simple traditional generic **sort** function:

```
template<typename Random_access_iterator, typename Compare = std::less>
void sort(Random_access_iterator first, Random_access_iterator last, Compare pred)
{
        // …
}
```

This style served C++ well for decades and is still part of the standard library. It takes a pair of iterators of some type indicating the range of elements [**first**:**last**) to be sorted and a function to compare elements (the sorting criteria). The sorting criteria is defaulted to less than.

This **sort** can be called for a variety of types of containers, a variety of element types, and a variety of sorting criteria. For example:

```
int a[] = { 3,1,4,2,6,9,0,-1};
vector<string> v = { "CPL", "BCPL", "C", "C++" };

sort(begin(v), end(v));                      // sort the vector
sort(begin(a), end(a));                      // sort the array
sort(begin(v), begin(v)+size(v)/2);          // sort the first half of the vector
sort(begin(v), end(v), greater<string>{});   // sort in ascenting order
sort(begin(a), end(a), [](int x, int y) { return abs(x)<abs(y); });  // sort absolute values
```

The **greater<string>{}** generates a standard-library function object that calls **<** when invoked. The **[](int x, int y) { return abs(a)<abs(y); }** in the last call of **sort** is a lambda expression, typically just called "a lambda". It is a notation for generating a function object. Such function objects (that is, objects with a call operator and possibly with a state) are extremely popular as arguments to generic functions.

This style of generic programming used here is very general, flexible, and traditional, but a bit verbose. Also, it is not type checked until late (at "template instantiation time"), implying awful error messages. By default, the implementation of **sort** is replicated for each argument type and then optimized for that type. Often, a compiler can merge common parts of implementations to avoid code bloat.

However, such interfaces violate the fundamental aim (and general principle) that interfaces should be precisely specified and hold enough information to allow checking (preferably at compile time and if not at run time). Only this century did we figure out how to achieve all three fundamental aims for generic programming support simultaneously.

5.2. Concepts

A concept is a compile-time predicate. That is, a function to be executed by the compiler, yielding a Boolean. It is mostly used to express requirements for the parameters of a template. Basically, we can ask a type if it has the desired properties. For example: “are you a range we can sort?”, “are you a number?” and “do you support the usual comparisons?”

A concept is often built from other concepts. For example, the **Sortable_range** used in

```
void sort(Sortable_range auto& r);
```

can be defined like this:

```
template<typename R>
concept Sortable_range =
        random_access_range<R>              // has begin()/end(), ++, [], +, …
        && sortable<iterator_t<R>>;         // can compare and swap elements
```

This says that a type **R** is a **Sortable_range** if it is a **random_access_range** and has an iterator type that is **sortable**. The **random_access_range** and **sortable** are **concept**s defined in the standard library.

However, this **Sortable_range** is not quite general enough to be useful as a library function. A concept can take several arguments; it is not just a type of types. For a general **sort**, we must be able to specify the comparison criteria as well as the range. That’s easily done:

```
template<typename R, class Pred = ranges::less>
concept Sortable_range =
        random_access_range<R>              // has begin()/end(), ++, [], +, …
        && sortable<iterator_t<R>, Pred>;   // compare elements using Pred
```

Not to complicate the simplest and most common cases, I defaulted the comparison function to less than. That’s conventional.

Given **Sortable_range**, we could define the range version of **sort** like this:

```
template<Sortable_range R, typename Pred = ranges::less>
void sort(R& r, Pred cmp = {});
```

and use it like this:

```
vector v = { 1,5,2,8,-1 };
```

```
sort2(v);                       // sort using <
sort2(v, ranges::greater{});    // sort using >
```

To specify a concept directly in terms of the language (as opposed in terms of other concepts), we use "use patterns" [GDR2006]. For example:

```
template<typename T, typename U = T>
concept equality_comparable = requires(T a, U b) {
        {a==b} -> Boolean;
        {a!=b} -> Boolean;
        {b==a} -> Boolean;
        {b!=a} -> Boolean;
}
```

The constructs in the **{…}** must be valid and return something that matches the concept specified after **->**. So here, the listed use-patterns (e.g**.**, **a==b**) must return something that can be used as a **bool**. Note that **equality_comparable** handles implicit conversions and **==** on different compatible types (e.g., integers and floating point values). That's essential for handling conventional C++ code.

Usually, as in the **sort** example, checking that a type matches a concept is done implicitly, but we can also be explicit using **static_assert**:

```
static_assert(equality_comparable<int,double>);   // succeeds
static_assert(equality_comparable<int>);          // succeeds (U is defaulted to int)
static_assert(equality_comparable<int,string>);   // fails
```

The **equality_comparable** concept is defined in the standard library. We don't have to define it ourselves, but it's a good example.

We have always had concepts. Every successful generic library has some form of concepts: in the designer's head, in the documentation, or in comments. Such concepts often represent fundamental concepts of an application area. For example:

- ***C/C++ built-in types***: arithmetic and floating [K&R1978]
- ***The C++ standard-library***: iterators, sequences, and containers
- ***Mathematics***: monad, group, ring, and field
- ***Graphs***: edges and vertices, graph, DAG, ...

C++20 didn't introduce the idea of concepts; it just added direct language support for concepts. A **concept** is a compile-time predicate. Programming using **concept**s is easier than not using them. However, like for every novel construct, we must learn to use it effectively together with other facilities.

5.3. Compile-time evaluation

A concept is an example of a compile-time function. In contemporary C++, any sufficiently simple function can be evaluated at compile time:

- **constexpr**: can be evaluated at compile time
- **consteval**: must be evaluated at compile time
- **concept**: evaluated at compile time, can take types as arguments

For example:

```
constexpr int isqrt(int n)        // evaluate at compile time for constant arguments
{
        int i = 1;
        while (i*i<n)
                ++i;
        return i-1;
}

constexpr int s2 = isqrt(1234);           // s2 is 35
```

We can do compile-time evaluation on both built-in and user-defined types. For example:

```
constexpr auto yule = weekday(December/24/2024);        // Tuesday
```

This gives us type-rich computation at compile time and saves us from writing error handlers for many functions.

To allow **consteval** and **constexpr** functions and **concept**s to be evaluated at compile time, they cannot

- have side effects
- access non-local mutable data
- have undefined behavior (UB)

However, they can use extensive facilities, incl. much of the standard library. Such functions are the C++ version of the idea of a pure function and a contemporary C++ compiler contains an almost complete C++ interpreter. Compile-time evaluation is also a boon to performance.

6. Guidelines and enforcement

Contemporary styles yield major benefits. However, old habits die hard. Familiarity is often mistaken for simplicity. Avoiding suboptimal techniques is difficult. Much confusing and outdated information is circulated on the Web and in teaching material. Modernizing existing code is hard and often expensive. Also, old code often offers outdated-style interfaces, thus encouraging the use of older styles of use. We need help to guide us to better design.

Stability/compatibility is a major feature. Also, given the billions of lines of C++ code, only gradual adoption of novel features and techniques is feasible. So, we can't change the language, but we can change the way it is used. People (quite reasonably) want a simpler C++, but also new features, and insist that their existing code must continue to run.

To help developers focus on effective use of contemporary C++ and avoid outdated "dark corners" of the language, sets of guidelines have been developed. Here, I focus on the C++ Core guidelines that I consider the most ambitious and best aligned with the ideals of C++ [GC].

A set of guidelines must represent a coherent philosophy of the language relative to a given use. They should codify the best contemporary C++ design and coding practices. My principal aim is a type-safe and resource-safe use of ISO standard C++. That is

- Every object is exclusively used according to its definition
- No resource is leaked

This encompasses what people refer to as memory safety and much more. It is not a new goal for C++ [BS1994]. Obviously, it cannot be achieved for every use of C++, but by now we have years of experience showing that it can be done for modern code, though so far enforcement has been incomplete.

A set of guidelines has strengths and weaknesses:

- It is available now (e.g., The C++ Core Guidelines [CG])
- Individual rules can be enforced or not
- Enforcement is incomplete

Building on guidelines, we need enforcement:

- A profile is an enforced coherent sets of guidelines rules [CG,BS2022]
- Being worked on in WG21 and elsewhere [BS2022b, HS2025]
- Are not yet available, except for experimental and partial versions [KR2019, Google2024, KV2024]

When thinking about C++, it is important to remember that C++ is not just a language but part of an ecosystem consisting of implementations, libraries, tools, teaching, and more. In particular, developers using C++ rely on facilities far beyond what you find in C++ textbooks and defined in the ISO standard.

6.1. Guidelines

Simple subsetting of C++ doesn't work. In particular, we need the low-level, tricky, close-to-the-hardware, error-prone, and expert-only features to implement higher-level facilities efficiently. The C++ Core Guidelines use a strategy known as subset-of-superset [BS2005]:

- ***First extend the language*** with a few library abstractions: use parts of the standard library and add a tiny library to make use of the guidelines convenient and efficient (the Guidelines Support Library, GSL).
- ***Next subset that superset*** by banning the use of low-level, inefficient, and error-prone features.

What we get is *"C++ on steroids":* Something simple, safe, flexible, and fast; rather than an impoverished subset or something relying on massive run-time checking. Nor do we create a language with novel and/or incompatible features. The result is 100% ISO standard C++. Messy, dangerous, low-level features can still be enabled and used where needed.

Different application domains have different needs and thus need different sets of guidelines, but initially the focus is on "The core or the C++ Core Guidelines." The rules we hope that everyone eventually could benefit from

- No uninitialized variables
- No range or nullptr violations
- No resource leaks
- No dangling pointers
- No type violations
- No invalidation

Two books describe C++ following these guidelines except when illustrating errors: "A tour of C++" for experienced programmers [BS2022] and "Programming: Principles and Practice using C++" for novices [BS2024]. Two more books explore aspects of the C++ Core Guidelines [JD2021; RG2022].

6.2. Example rule: Don't subscript pointers

A pointer doesn't have the associated information needed to allow range checking. However, range checking is a must for memory safety and also for type safety because we cannot allow application code to read or overwrite objects beyond the range of the objects pointed to. Instead, we must use abstractions with enough information to range check, such as arrays, **vector**s, and **span**s.

Consider a common style: a pointer plus an integer supposedly indicating the number of elements pointed to:

```
void f(int* p, int n)
{
        for (int i = 0; i<n; i++)
                do_something_with(p[n]);
}

int a[100];
// …
f(a,100);       // OK? (depends on the meaning of n in the called function)
f(a,1000);      // likely disaster
```

This is a very simple example using an array to show the size. Since the size is present, checking at the point of call is possible (though hardly ever done) and typically a (pointer,integer) pair is passed through a longer call chain making verification difficult or impossible.

The solution to this problem is to tie the size firmly to the pointer (like in **Vector**; §3.1). That's what a **span** does:

```
void f(span<int> a)     // a span holds a pointer and the number of elements pointed to
{
        for (int& x: s) // now we can use a range-for
                do_something_with(x);
}

int a[100];
// …
f(a);           // the type and element count are deduced
f({a,1000});    // asking for trouble, but marked syntactically and easily checkable
```

Using **span i**s a good example of "Make simple things simple" principle. Code using it is simpler than the "old style": shorter, safer, and often faster.

The **span** type was introduced in the Core Guidelines support library as a range-checked type. Unfortunately, when it was added to standard library, the guarantee for range checking was removed. Obviously, a profile (§6.4) enforcing this rule must range check. Every major C++ implementation has ways to ensure this (e.g., Clang standard library hardening [KV2024], Google Spatial safety [Google2024], and Microsoft's debug mode and lifetime profile [KR2019]). Unfortunately, there isn't yet a standard and portable way of requiring it.

6.3. Example rule: Don't use an invalidated pointer

Some containers, notably **vector**, can relocate their elements. If someone outside the container obtains a pointer to an element and uses it after relocation, disaster can happen. Consider:

```
void f(vector<int>& vi)
{
    vi.push_back(9);        // may relocate vi's elements
}

void g()
{
    vector<int> vi { 1,2 };
    auto p = vi.begin();    // point to first element of vi
    f(vi);
    *p = 7;                 // error: p is invalid
}
```

Given appropriate rules for the use of C++ (§6.1), local static analysis can prevent invalidation. In fact, implementations of the Core Guidelines lifetime checks have done that since 2019 [KR2019]. Prevention of invalidation and the use of dangling pointers in general is completely static (compile time). No run-time checking is involved beyond checking for the **nullptr**. A pointer the passed the lifetime check is either valid or the **nullptr**.

This is not the place for a detailed description of how this analysis is done. For that see [BS2015; HS2019; BS2024]. However, here is an outline of the model:

The rules apply to every entity that directly point to an object, such as pointers, resource-management pointers, references, and containers of pointers. Examples are an **int***, an **int&**, a **vector<int*>**, a **unique_ptr<int>**, a **jthread** holding an **int***, and a lambda that has captured an **int** by reference.

- Ban use after **delete** (obviously) and rely on RAII (§3).
- Don't allow a pointer to escape the scope of what it points to. This means that a pointer can be returned from a function only if it points to something static, points to something on the free store (aka heap and dynamic memory), or was passed in as an argument.
- Assume that a function (e.g., **vector::push_back()**) that take non-**const** arguments invalidates. If a pointer to one of its elements has been taken, we ban calls to it. Functions taking only **const** arguments cannot invalidate, and to avoid massive false positives and preserving local analysis, we can annotate function declarations with **[[profiles::non_invalidating]]**. This annotation can be validated when we see the function's definition. Thus, it is a safe annotation rather than a "trust me" annotation.

Naturally, there are many details to address, but they have been tried out in experimental as well as currently shipping implementations.

6.4. Enforcement: Profiles

Guidelines are fine and useful, but it is essentially impossible to follow them consistently in a large code base. Thus, enforcement is essential. Enforcement of rules preventing missing initialization, range errors, **nullptr** dereferences, and the use of dangling pointers is currently available and have been demonstrated to be affordable in large code bases [KR2019, Google2024, KV2024].

However, key foundational rules must be standard – part of the definition of C++ – with a standard way of requesting them in code to enable interoperability between code developed by different organizations, code running on multiple platforms, and code used in teaching.

We call an enforced coherent set of guideline rules providing a guarantee a "profile." As currently planned for the standard, the initial set of profiles (based on Core Guidelines profiles as used for years) are [HS2025; CG]:

- **type** – every object initialized; no casts; no unions
- **lifetime** – no access through dangling pointers; pointer dereference checked for **nullptr**; no explicit **new**/**delete**
- **bounds** – all subscriping is range checked; no pointer arithmetic.
- **arithmetic** – no overflow or underflow; no value-changing signed/unsigned conversions

This is essentially "the core of the core" described in §6.1. More profiles will follow given time and experimentation [BS2024b]. For example:

- **algorithms** – all ranges, no dereferences of **end()** iterators
- **concurrency** – eliminate deadlocks and data races (hard to do)
- **RAII** – every resource owned by a handle (not just resources managed with **new**/**delete**).

Not all profiles will be ISO standard. I expect to see profiles defined for specific application areas, e.g., for animation, flight software, and scientific computation. This reflects the fact that "safety" isn't one thing. Different domains need enforcement of different sets of rules to meet their requirements.

Enforcement is primarily static (compile-time) but a few important checks must be run-time, (e.g., subscripting and **nullptr** dereferencing).

A profile must be explicitly requested for a translation unit. For example,

```
[[profiles::enforce(type)]]        // no casts or uninitialized objects in this TU
```

Where necessary, a profile can be suppressed for a statement (including compound statements) where needed. For example:

```
[[profiles::suppress(lifetime)]]    // no  dangling pointer checks for the next statement
this->succ = this->succ->succ;
```

The need to suppress verification of guarantees is primarily for the implementation of the abstractions needed to provide guarantees (e.g., **span**, **vector**, and **string_view**), to guarantee range checking, and to directly access hardware. Because C++ needs to manipulate hardware directly, we cannot "outsource" the implementation of fundamental abstractions to some other language. Nor – because of its wide range of applications and multiple independent implementations – can we simply leave the implementation of all foundational abstractions (e.g., all abstractions involving linked structures) to the compiler.

Obviously, suppression should be minimized. However, it is localized and easily searched for. Importantly, suppression is of a specific guarantee; it is not all or nothing.

Most safety requirements are not mere programming language properties but are system requirements involving tools and techniques beyond the scope of what a programming language can offer. Profiles aim to provide refined language guarantees, but they are not – cannot be – a substitute for the many tools and techniques that people today use to provide reliable systems. Having multiple profiles reflects the reality that no one size fits all.

7. The Future

I am reluctant to make predictions about the future, partly because that's inherently hazardous, and in particular because the definition of C++ is controlled by a huge ISO standards committee operating on consensus. Last I checked, the membership list had 527 entries. That indicates enthusiasm, wide interest, and provides broad expertise, but it is not ideal for programming language design and ISO rules for making a standard cannot be dramatically modified to cope. Among other subjects, there is work in progress on

- A general model for asynchronous computing [LB2024]
- Static reflection [WF2024]
- SIMD [MK2024]
- A contract system [JB2024]
- Functional-programming style pattern matching [HS2024; MP2024]
- A general unit system (e.g., the SI system) [MP2024]

Experimental versions of all of these are available.

One serious concern is how to integrate diverse ideas into a coherent whole. Language design involves making decisions in a space where not all relevant factors can be known, and where accepted results cannot be significantly changed for decades. That differs from most software product development and most computer science academic pursuits. The fact that almost all language design efforts over the decades have failed demonstrates the seriousness of this problem.

8. Summary

C++ was designed to evolve. When I started, not only didn't I have the resources to design and implement my ideal language, but I also understood that I needed the feedback from use to turn my ideals into practical reality. And evolve it did while staying true to its fundamental aims [BS1994]. Contemporary C++ (C++23) is a much better approximation to the ideals than any earlier version, including support for better code quality, type safety, expressive power, performance, and for a much wider range of application areas.

However, the evolutionary approach caused some serious problems. Many people got stuck with an outdated view of what C++ is. Today, we still see endless mentions of the mythical language C/C++, usually implying a view of C++ as a minor extension of C embodying all the worst aspects of C together with grotesque misuses of complex C++ features. Other sources describe C++ as a failed attempt to design Java. Also, tool support in areas such as package management and build systems have lagged because of a community focus on older styles of use.

The C++ model can be summarized as

- Static type system
  - Equal support for built-in types and user-defined types
  - Value and reference semantics
- Systematic and general resource management (RAII)
- Efficient object-oriented programming
- Flexible and efficient generic programming
- Compile-time programming
- Direct use of machine and operating system resources
- Concurrency support through libraries (supported by intrinsics)
- Eliminate the C preprocessor

The C++ language and standard library are the concrete expression of this model and a critical part of the ecosystems used to develop software. The value of a programming language is in the quality of its applications.

9. References

- [CG] B. Stroustrup and H. Sutter (editors): C++ Core Guidelines.
- [DE2021] D. Engert: A (Short) Tour of C++ Modules . CppCon 2021.
- [DE2022] D. Engert: Contemporary C++ in Action. CppCon 2022.
- [GCC2024] GCC exception performance .
- [GDR2006] G. Dos Reis and B. Stroustrup: Specifying C++ Concepts. POPL06. 2006.
- [Google2024] A. Rebert et al: Retrofitting spacial safety to hundreds of millions of lines of C++. 2024.
- [HS2019] H. Sutter: Lifetime safety: Preventing common dangling. WG21 P1179. 2019-11-22.
- [HS2024] H. Sutter: Pattern matching using is and as. WG P392R3. 2024.
- [HS2025] H. Sutter: Core safety profiles for C++26. WG21 D3081R1. 2025.
- [JB2024] J. Berne, T. Doumler, and A. Krzemieński: Contracts for C++. P2900R9. 2024.
- [JD2021] J. Davidson and K. Gregory Beautiful C++: 30 Core Guidelines for Writing Clean, Safe, and Fast Code. 2021. ISBN 978-0137647842.
- [KE2024] K. Estell: C++ Exceptions for Smaller Firmware. CppCon 2024.
- [K&R1978] B.W. Kernighan and D.M. Ritchie: The C Programming Language. Prentice-Hall. 1978. ISBN 01-13-110163-3.
- [KV2024] K. Varlamov and L. Dionne: Standard library hardening. WG21 P3471R0. 2024.
- [KR2019] K. Reed: Lifetime Profile Update in Visual Studio 2019 Preview 2. 2019.
- [LB2024] L. Baker et al: A plan for std::execution for C++26. WG21 P3109R0. [MK2024] M. Kretz: std::simd — data-parallel types. WG21 P21928R9. 2024.
- [MP2024] M. Park: Pattern Matching: match Expression. WG21 P2688R2. 2024.
- [MP2024] M. Pusz et al: Quantities and units library. WG21 P3045R2. 2024.
- [RG2022] R. Grimm: C++ Core Guidelines Explained. Addison-Wesley. 2022. ISBN 978-0136875673.
- [BS1982] B. Stroustrup: Classes: An Abstract Data Type Facility for the C Language. SIGPLAN Notices, January 1982.
- [BS1993] B. Stroustrup: A History of C++: 1979-1991. ACM SIGPLAN. March 1993.
- [BS1994] B. Stroustrup: The Design and Evolution of C++. Addison Wesley. ISBN 0-201-54330-3. 1994.
- [BS2005] B. Stroustrup: A rationale for semantically enhanced library languages. LCSD05. October 2005.
- [BS2007] B. Stroustrup: Evolving a language in and for the real world: C++ 1991-2006. ACM SIGPLAN. June 2007.
- [BS2015] B. Stroustrup, H. Sutter, and G. Dos Reis: A brief introduction to C++'s model for type- and resource-safety. Isocpp.org. October 2015. Revised December 2015.

- [BS2020] B. Stroustrup: Thriving in a Crowded and Changing World: C++ 2006–2020. ACM SIGPLAN. June 2020.
- [BS2021] B. Stroustrup: Minimal module support for the standard library. WG21 P2412r0. 2021.
- [BS2022] B. Stroustrup: A Tour of C++ (3rd Edition). Addison-Wesley. 2022. ISBN 978-0-13-681648-5.
- [BS2022b] B. Stroustrup and G. Dos Reis: Design Alternatives for Type-and-Resource Safe C++. WG21 P2687R0. 2022.
- [BS2024] B. Stroustrup: Programming: Principle and Practice using C++. Addison-Wesley. 2024. ISBN 978-0-13-830868-1.
- [BS2024b] B. Stroustrup: A framework for Profiles development. WG21 P3274R0. 2024.
- [BS2024c] B. Stroustrup: Profile invalidation - eliminating dangling pointers WG21 P3346R0.
- [WF2024] W. Childers et al: Reflection for C++26 . WG21 P2996R6. 2024.

Acknowledgements

Thanks to the readers of drafts of this paper: Joseph Canero, James Cusick, Vincent Lextrait, Christoff Meerwald, Nicholas Stroustrup, Andreas Weiss, J.C. van Winkel, Michael Wong, and of course the many contributors to the C++ Core Guidelines and the evolving proposal for ISO-standard profiles.