Technical Note. From C++98 to C++2x

Konstantin Burlachenko

King Abdullah University of Science and Technology, Thuwal, Saudi Arabia.

Editors

- Vadim Sofin ex-HUAWEI / ex-Yandex.
- Mikhail Filimonov from NVIDIA.
- Dmytro Ovdiienko Principal Software Engineer *Low Latency Systems* from Quiet Light Trading.

Revision Update: May 1, 2025

Historical Note: The original title "*Technical Note. From C++1998 to C++2020*" in the update from March 2024 has been changed due to the additional appendix to cover some language and library features of C++ 2023.

Table of Contents

Table of contents generated with markdown-toc

Introduction

On that technical note, we would like to share complete information regarding the C programming language and all primary C++ programming language standards: C++03/98, C++11, C++14, C++17, C++20, and C++23. If you do not know C++, this note is less likely for you because it contains subtle technical details for people who are at least a bit familiar with it. Here, "know" has a weak sense. We have also tried to appeal in that note to people with a not-so-big background in C++.

Do not get us wrong. If you have never seen the C++ language to obtain knowledge, we recommend first dedicating some time to reading original books written by the original author of the C++ Language Bjarne Stroustrup. It would only be more effective for you. In recent years, Bjarne Stroustrup has made a lot of effort by providing easy-to-read books such as "Principles and Practice Using C++" and "A Tour of C++ (Second Edition)". We highly recommend that those for whom this language is new first read any of those books.

If you're unsure whether you should learn C++ or not, then maybe the example presented in section *Why learn C++ if I know Python (Toy Example)* of this document will bring some consideration to your mind. C++ is complex, but currently, it's one of the fastest (in terms of execution speed in CPU) high-level, general-purpose programming languages in the world. It can be observed from comparison tests such as benchmarksgame-team.pages.debian.net/benchmarksgame and checking the language in which compute-demanding applications in your domain have been written. In our experience, in most cases, it will be C or C++.

Sometimes, you must write software for a software platform (Java Virtual Machine, JavaScript Engine, Python interpreter). This is the case, for instance, when you can not execute real code in a target computing machine for some reason. In such circumstances, the de facto standard can be another programming language, not C and not C++ at all, and not a dialect of C or C++. Analyzing when it is good or bad to limit users from using C++ or any compiled language is out of the scope of this technical note.

It is important to note that there is a notion of a programming language (in a very strong sense) that converts algorithms into the language of a computing machine. If Language always requires a software platform to operate, it is not a programming language according to this possible definition (See https://www.stroustrup.com/bs_faq.html#Java or https://www.stroustrup.com/bs_faq.html#Csharp for a discussion about this).

This note is mainly based on materials from the references section and personal experience. We think that information can be helpful for three categories of people:

- People who want to refresh or go deep into several classical language constructions of C++
- People who want to obtain a pretty in-depth overview of new features from C++11/14/17/20/23
- Elements of this note can be helpful for people who need to support (legacy) C++03 or C99 code base

Finally, we welcome anybody who wants to make this note cleaner. We appreciate the style of *Language Lawyer* and *Practical Applicability*. Still, we don't want to have any of the extremes of both types, because neither of them in extreme limits helps create new algorithms and new software.

Prepare Environment

To reproduce code snippets you need to have a C++ IDE (e.g. Visual Studio, QtCreator, Xcode, CLion) or command line environment in which you will launch compiler and linker to compile and link code snippets. Currently, there are plenty of versions of the C++ programming language, and you will need to provide compiler information about the version of the language standard that you're going to use.

It can be accomplished by providing the compiler with special flags for the compiler program. If you are using an Integrated Development Environment (IDE), it can be done with extra help from its Graphical User Interface (GUI), or you should find a way to adjust compiler flags in it manually in IDE-dependent way:

- **Visual Studio/MSVC.** Specify `/std:c++20` or `/std:c++latest` for MSVC compiler. Please use *MSVC 19.30* or higher for code snippets in the text. Visual Studio 2022 Community Edition is distributed with *MSVC 19.32* at the moment of writing this text. The MSVC version list can be obtained from MSVC Wiki.

- **GCC**. Specify `-x c++ --std=c++20` if you're using GNU Compiler Collection (GCC). Please use *GCC 10.1* or higher for code snippets in the text. The GCC version list is available GCC-Releases.

- **CLANG/LLVM**. Specify `--std=c++20` if you're using CLang. Please use *Clang 10.0.0* or higher. The Clang/LLVM version list is available LLVM-Relesases.

Support of Various Features by Compilers Vendors

Compiler vendors are hard at work to catch up with all the new features, which have started to appear every three years since 2011.

You can keep track of which compiler supports which features of C++11/14/17/20/23 based on this table: https://en.cppreference.com/w/cpp/compiler_support

An alternative and in fact standardized way, starting from C++2020, to check the presence of some features is based on using feature test macro. The list of available macros is available here: https://en.cppreference.com/w/cpp/feature_test

Glossary

C/C++. By C/C++, we mean C or C++ programming languages. Formally, these are two different languages.

A Shallow Copy. A shallow copy contains copies of all members of an object one by one. If the copied members are pointers to dynamic memory, then only pointers by themselves are copied. Objects to which data pointers refer are not taken into consideration during shallow copying.

A Deep Copy. A deep copy copies all dynamic memory objects referred to by any pointer members and data fields/members of the object as well.

Upcast. Casting object to its base class.

Downcast. Casting object to the derived class from the base class.

Function Signature. The function name and the parameter list with type information are called the signature of a function.

Function Prototype (function declaration). The language statement that describes a function in a form that is sufficient for the compiler to be able to compile calls to it.

Template Type Parameter. Type placeholder used in class or function template, typically denoted by `T`. Example:

```
template <class T>
class MyClass{};
```

Template Type Argument. The type assigned to a template type parameter `T` during template class instantiation or template function instantiation. Therefore, it's a type with which you specify the template to be instantiated.

Function Object (or functor). Object of a class that overloads the function call operator. Example:

```
class Area {
public:
  double operator()(double w, double h) const {
    return w * h;
  }
};
```

Pure Virtual Function. The purpose of the virtual function is to enable the derived class versions of the function to be called polymorphically (in different ways). The purpose of a *pure* virtual function is to have polymorphic behavior in a case when the implementation of that function in the base class is absent.

Example:

```
class Shape {
public:
   virtual double area() const = 0;
};
```

Abstract Class. A class that contains at least *one* pure virtual function.

LValue (expression). An `LValue` evaluates during compile time to some persistent value which conceptually has an address in memory where you can store something. Informally, that is something to the left of the operator equals.

LValues is an attribute to describe expressions in the Language, not value, even "value" is presented in the term.

RValue (expression). An RValue evaluates a result that is stored only transiently. An expression from which the address cannot be taken conceptually. This is something that, at least in principle, can be encoded in the code of generated instructions for the processor. In this way, the value is not stored as a memory object, but encoded in the instruction itself. Rvalue is an attribute to describe *expressions* in the Language.

In 99% of cases, these are unnamed temporary variables. A good counterexample of something that is an *RValue*, but it has the name is `this.`

> Unfortunately, starting from C++11, the *object type* and *reference type* do not match each other due to a more complicated picture with expressions(values) and references.

XValue (expression). Objects in memory that would be destroyed very soon. It's an object for which it is reasonable to use move semantics to take data via the `T&&` notation from C++11. The letter "x" in **X**Value stands for e**X**piring value.

LValue Reference (for all C++ versions). An LValue reference is an alias for another variable. LValue object may be bound to the LValue reference through the following syntax, which essentially creates one more name (alias) to a variable:

```
X& x = obj; // X is the datatype of obj
```

RValue Reference (only for C++98/03). In C++98/03, it is (i) a usual const regular reference to a temporary object; (ii) an expression that potentially is an lvalue object, but which is used from the right-hand side of the operator `=` and similar assignment operators.

RValue Reference (starting from C++11). The goal of an RValue reference is to have a moving candidate for functions like `void f(T&&)`. In practice, RValue reference is either:

- A reference to an object that will soon be deleted (xvalue expression)
- Explicitly unconditionally cast through `std::move` to an RValue reference. The `std::move` brings an object after *moving* to a valid, but undefined state.

> Reusing an object after "moving from it" is in fact *legal* and *valid*. In one of the talks in CppCon Nicolai M. Josuttis, a member of the C++ Standard Committee, *explicitly highlighted it*. In that case, if you want to use the object from which you moved out of the state, you should reinitialize the object using the class API or the logic behind the class.

> What was known in C++03/98 as *RValue Reference* in context of applying for non-temporary objects starting from C++11 has been renamed into *Const LValue Reference*.

Token. In the terminology of Programming Languages, tokens are separate words of a program text. One easy case is when such words (tokens) are split between each other by spaces. A harder case is to identify tokens from a sequence of characters in a situation when there are no whitespaces.

Pure Function. The pure function is a type of function (used in the case of using `constexpr` and `consteval` modifiers) in C++ when you ask or you are required to have a *C++ function which coincides with a mathematical function in meaning*. Specifically, we can name the function as **pure** if the following holds:

1. Function produces the same output if called with the same arguments in the future.
2. There are no side effects in the program environment.
3. The function does not change the state of the program.

> If you have a digital design background, this type of logic is named Combinational Logic (CL), which is stateless.

Incomplete Type. In C and C++, there are two cases when you can use a type with an undefined size:

- Creating a pointer for a type
- Creating an alias name via a *typedef* (in C and C++) or via *using* (C++) keywords.

Essentially, it is possible because the size is not required for a declaration of a pointer and an alias name. [2, p.150]

```
class A;        // Class declaration, but no definition

typedef A B;    // Typedef for C++98/03

using BB = A;   // Type Alias (also known as smart typedefs) for C++11

A* ptr;
```

Statement Block. Several C++ or C language statements organized by a pair of curly braces {, }.

CV qualifiers. During reading standards or another form of language rules, you will at some moment be faced with cv qualifier term. It means *constant* and *volatile* type qualifiers. The *volatile* is a type qualifier that denotes that that object can alter its value not due to C++ language, but for other system reasons that can not be captured by the language per se.

Motivation

Both C and C++ programming language represents a pretty thin abstraction over the underlying hardware. The software level below C and C++ is the Assembly Language for your computing device. Why performance is critical is excellently motivated by Prof. Charles E. Leiserson from MIT, in his undergraduate course about Algorithms and Data structures. In the first lecture. Prof. Charles E. Leiserson in 2005 (MIT Introduction to Algorithms 2005, Lecture 1) highlighted that there are a lot of things that are more important than performance:

- Modularity
- Correctness
- Maintainability
- Functionality
- Robustness
- User-friendliness
- Programmer time
- Simplicity
- Extensibility
- Reliability
- Security
- Scalability

The **Performance** is the goal in case of having real-time requirements for the software. If the software is not fast enough, it's just not a choice to use or buy it in such circumstances.

But in fact, if think deeply then in most cases the performance is due to Prof. Charles E. Leiserson is the currency (money) under which it's possible to buy other features from the list above because these features are *not coming for free*.

Nowadays, in 2024 due to Tobex Index May 2024, the interpretable scripting language Python is the most popular in the world. Interestingly, Python was originally designed only as a replacement for Bash. That has been described in that Blog Post written by the author of the Python scripting Language, Guido van Rossum:

> "...My original motivation for creating Python was the perceived need for a higher-level language in the Amoeba project. I realized that the development of system administration utilities in C was taking too long. Moreover, doing these in the Bourne shell wouldn't work for a variety of reasons. The most important one was that as a distributed micro-kernel system with a radically new design, Amoeba's primitive operations were very different (and finer-grain) than the traditional primitive operations

> available in the Bourne shell. So there was a need for a language that would "bridge the gap between C and the shell..." - Guido van Rossum.

It is not a secret anymore that today people try to apply Python beyond launching scripts, but creating other user space applications. It may be a choice to use Python in the case of:

- (a) The Underlying Algorithms that you need to use are implemented in C++
- (b) The Underlying Algorithms that you need to use are implemented in Hardware
- (c) The algorithms are available via Python bindings with suitable interfaces
- (d) The overhead of the Python Interpreter is negligible
- (e) There is a big part of the system that has already been implemented in Python

We believe the primary reason for Python's popularity is its fast learning curve, which can be measured in just three days (using only the language, without external libraries, frameworks, or middleware). At the same time, it's impossible to learn C++ in 3 days. We think the C++ community should think about it for its survival.

But any interpretable languages are not a choice when actual time matters, or subtle control over the memory matters, or any memory/compute inside any device connected to the computer matters, even for programming only in user space.

Downsides of Interpretable Languages

Probably, it's worthwhile to highlight some downsides of interpretable languages and articulate them exactly:

1. The interpreter parses the program's text (source code) line by line (that is represented in text form or extremely high-level instructions). It is highly inefficient from the start, even to execute such high-level code. As a consequence, Interpretable languages provide algorithms in practice that can be even up to 50'000 times slower in computing than highly optimized C/C++/ASM code (and such examples exist). The interpreter is the worst possible choice for execution time from all three possible choices for converting source code into software: (Interpreter, Just In Time compiler, and Compiler).

   > For a concrete example, please look at Lecture 1 from 6-172. Performance Engineering of Software Systems at MIT with Prof. Charles E. Leiserson. The overview of that course is also available here: About Performance Engineering course 6.172 at MIT.

2. Interpretable languages do not provide subtle interfaces to Operating Systems such as POSIX API, Windows API or other OS-dependent APIs. It provides bindings for API that the team that developed the interpreter had time to finish, and they are provided in a highly simplified form.

3. To some extent, interpreters provide portability in the source code for user space applications. Still, it comes with the cost of reducing the number of possible calls to OS. Creating portability at the source code level between different OS is a big thing, and people thought about that in the past. The problem of understanding led to the creation of POSIX, which was a way to provide portability between different OS via the standardization of many everyday routines for the OS API itself. If the goal is portability between different OS, the more correct approach is to solve it via the standardization of the API to the OS. Creating extra software layers, especially in the form of interpreters, which can not be eliminated during runtime, is a suboptimal decision if speed or memory matters.

4. During work with interpretable languages, you don't have a real interface to work with the devices' memory inside the computer and devices in general in all possible ways provided by the OS. You do not

even have enough tools to precisely handle just the usual *virtual memory* in your **own process**.

5. The interpreter, as a computer program, adds an extra level of abstraction. The standard implementation of the Python interpreter is CPython (https://github.com/python/cpython). It is called CPython because it has been implemented in C. Such software as an interpreter improves the time for completing the project from a social point of view, but the implementation of your algorithms, and underlying system can be suboptimal from an execution time standpoint (of course provided you have the skills and time to implement, debug, profile the functionality on your own).

6. The absence of a compiler has *pros* - you do not spend time on a compilation, but there are *cons* - now, the compiler will not tell you about errors in the code because there is no compiler. And not only this, you can not adjust the compilation for your needs.

7. Uncontrollable memory allocations in a program that should execute and live in the OS for a long time, and during runtime require extra memory allocation, which may lead to memory fragmentation and other memory problems. In Python, you don't have control over the memory in your application. These uncontrollable memory allocations can happen in the Python runtime or inside external C or C++ libraries on which Python depends.

8. Compiler optimization tricks, such as code inlining, are out of the scope of any interpretable language because to perform such optimization, you should have a compiler. The elimination of the compiler stage will make this optimization and other code optimization executed during compile-time impossible for user-defined algorithms, literally due to the absence of the compiler.

9. During creating multithread implementation, you should be careful about memory fences (memory barriers or memory fences are used to enforce ordering constraints of executed instruction in a superscalar processor before and after memory barrier/fence), synchronization, data races, atomic operations, absence of storing some objects in registers via volatile qualifier (which is absent in Python language). In reality, the implementation of interpreters is typically highly leveraged into existing C or C++ libraries because creating actual modules of functionality in an interpreter by itself is not effective. However, it is not true that all C and C++ libraries are thread-safe. And so, creating a true multithreading environment inside an interpreter can be tricky. If you want to learn more about how Concurrency in Python is implemented (and want to know more about Global Interpreter Lock (GIL)), we recommend talks by one Python enthusiast, David Beazley: An Introduction to Python Concurrency, David Beazley. Do not get us wrong. Developers of the Python interpreter did their best, but the problem was not so easy in the first place. Once the Python interpreter resolves it, then at least you should observe speedup from this compute-bound example:

```
#!/usr/bin/env python3
# single_count.py (Example has been taken from https://realpython.com/python-gil/)

import time
from threading import Thread

COUNT = 50000000

def countdown(n):
    while n>0:
        n -= 1
```

```
start = time.time()
countdown(COUNT)
end = time.time()

print('Time taken in seconds -', end - start)
```

```
#!/usr/bin/env python3
# mth_count.py (Example has been taken from https://realpython.com/python-gil/)
import time
from threading import Thread

COUNT = 50000000

def countdown(n):
    while n>0:
        n -= 1

t1 = Thread(target=countdown, args=(COUNT//2,))
t2 = Thread(target=countdown, args=(COUNT//2,))

start = time.time()
t1.start()
t2.start()
t1.join()
t2.join()
end = time.time()

print('Time taken in seconds -', end - start)
```

10. Garbage Collector (GC) brings various limitations to any programming language. For example, GC disallows any pointer arithmetic and execution of GC is not free. (For details, please look at Lecture 11 from 6-172. Performance Engineering of Software Systems at MIT). Therefore the adoption of the idea of GC in the language has consequences.

11. Due to high abstraction, Interpretable Languages violate memory locality principles because almost every object is allocated on the heap. Memory locality is implemented typically in two forms. First, you can allocate objects on the stack and use stack memory as a memory placeholder. Second, another form of memory locality is when you have a C structure and nearby fields, located nearby in memory. Memory Locality is an essential principle because on that principle all memory caches in all levels of various memory storage are working inside modern computing devices. High-level abstraction languages (such as Python) almost ignore this aspect of reality.

12. Processors have a limited number of registers in their front end (available when you write an Assembly program or compile and generate an Assembly program for you). Correctly performing register allocation from the processor backend is not so trivial. If you have too many objects with too many wrappers around them, the useful load for a real final computer device is smaller and degrades. The

relative term from the Communication Community for this phenomenon is *goodput*, which is the throughput from which all service information has been removed. So, extra wrappers around even the simplest objects are not good for actual performance.

13. There is no way to use special registers or special instructions of the processor from typical interpretable languages (Bash, Python) directly.

14. The modern CPU devices (after 1980) and GPU compute devices are pretty complicated pipelined devices with different Functional Units (FU). Also, such devices have L1, L2, and L3 Data and Instruction Caches. To utilize these Caches, the program should execute instructions in ISA for the CPU. Unfortunately, if you execute the interpreter, caches will likely hold data and instructions of the Interpreter itself.

The interpretable language is excellent for prototyping. But any interpreter, any user space algorithm in it, can be beaten already by C++/ASM implementation both in used memory and compute time on the same hardware. At least be aware of that.

Downsides of C and C++

1. C++ is pretty complex if considering all language details. That aspect is not suitable for spreading the language in society fast.

2. The powerful expressivity of C++ is a technical power. At the same time, it's a weakness in obtaining new adepts.

   > Without new adepts, any concept will die.

3. Due to the high entry level for C++ in Academia, high momentum belongs to Python, not to C++ at all. C++ is used only when necessary. Examples:

   - You need to work with the hardware directly
   - You need to have algorithm/mathematical model that operates in real time
   - You need to be careful in terms of consumed memory

4. The speed of development of the prototype is faster in Python for typical user space applications. Fast iteration is essential in a lot of cases because research and innovation deal with uncertainty all the time in various forms.

5. C++, to some extent, forces you to be aware of the hardware level. It's not clear whether it is good or bad:

   - On one side, when you want to try an idea, interpretable language provides a fast way to do that if you are not going to touch hardware features or operating system features, and you will not measure actual time or bytes in any form.
   - On the other hand, widespread usage of interpretable languages will lead to situations in which many people will not know how the computer works. You will lose the ability to distinguish a big lie from a small lie and truth in the context of computing machines.

6. The next problem that people with Python backgrounds face is the need to install Toolchains and build systems. In fact, if you are new to these languages, then you even have to learn the syntax of build systems, e.g., CMake. With a scripting language, such a thing is absent in the first place. Once you have

the interpreter installed, and you install needed libraries via the package manager in scripting languages, you just launch the script if it implements standalone functionality (conceptually).

People continue to predict that C++ will die. It's an ongoing and fun *three-decade* process, but it is not happening. It seems that the fundamental things of the language make it *immortal*, even though the language tends to be more complex.

Deep Principles of the Language

The Language started as a project in Bell Labs in 1979 ([3], https://www.stroustrup.com/bs_faq.html#invention. The principles of C++ language, which B.Stroustrup put into the Language, were documented between 1981 and 1991. According to B.Stroustroup, they existed even before the decision of standardization that took place in 1989. More importantly, the principles of the Language, even in 2024, are still inside it, and they are the heart of the Language ([3]):

1. No implicit violation of the static type system.
2. Provide the same good support for user-defined types as the implementation provides for its built-in types.
3. The locality of memory access patterns for variables and arrays is suitable for hardware in the long term, and the language should support it.
4. Two Zero-Overhead principles:
    - a. What you don't use, you should not pay for.
    - b. If something is built into the Language itself, it's impossible to write it better by hand.

Some language decisions due to B.Stroustoup:

- "*C++ does not have a universal class Object. It's so because, in C++, we don't need one: generic programming provides statically type-safe alternatives in most cases. Also, there is no valid universal class; in fact, using a universal base class implies the cost.*"
- "*Templates are not Generics (from C# or Java). Generics are primarily syntactic sugar for abstract classes. With generics (whether Java or C# generics), you program with precisely defined interfaces, typically pay the cost of virtual function calls and/or dynamic casts to use arguments.*"

Why learn C++ if I know Python (Toy Example)

Sometimes, while making programs in Python, you need to write programs directly in Python, not only call external C++ libraries from it. Possible reasons why you need to implement the algorithm in Python (without leveraging external libraries):

- The algorithm is short and suitable for CPU.
- Library does not exist, or Library exists but does not provide Python bindings.
- The library does not provide enough configuration, and you need it.
- You need to change something fundamental inside the C++ Library. You don't know C++, and due to a lack of knowledge or because the code is too complicated, you use Python.

Creating a CPU-effective algorithm in Python is difficult when wall clock time matters. As a concrete example, let's compare the wall clock time of two programs written in C++11 and Python3 under the following assumptions:

- Both programs use single-core CPU
- The C++ program does not use any special optimization techniques. It's a usual C++ code.

The test compares the wall clock time of the following:

1. Python with native Python lists
2. Python implementation with NumPy arrays
3. Cython(a programming language that mixes C and Python)
4. Flat C (or C++) arrays.

The task is performing setup elements in the array with 10M elements as an arithmetic sequence, performing their summation, and converting the result to `double` (fp64). Our OS is Ubuntu 18.04.6, x86_64.

1. Native Python implementation

```
#!/usr/bin/env python3
# Plain Python Code
import time

start = time.time()
a = [0] * 10*1000*1000
s = 0.0

for i in range(10*1000*1000):
    a[i] = i
    s += float(a[i])
end = time.time()

print(f"Processing {len(a)/1000000}M elements takes: ", (end - start) * 1000.0, "
milliseconds")
print(str.format("Sum is {0:g}" , s))
```

Output for Python 3.6.9:

```
Processing 10.0M elements takes:  2193.230390548706 ms
Sum is 5e+13
```

2. Python implementation using numpy

```
#!/usr/bin/env python3
# Python Code that leverages Numpy Library 1.22.4
# pip install numpy

import time
```

```
import numpy as np

start = time.time()

a = np.zeros(10*1000*1000, dtype=np.int32)
for i in range(10*1000*1000):
    a[i] = i
s = a.sum(dtype=np.double)
end = time.time()

print(f"Processing {len(a)/1000000}M elements takes: ", (end - start) * 1000.0, "
milliseconds")
print(str.format("Sum is {0:g}" , s))
```

Output for Python 3.6.9:

```
Processing 10.0M elements takes:  1051.522970199585  milliseconds
Sum is 5e+13
```

3. Cython implementation

```
#!/usr/bin/env python
# filename: setup.py
# Cython version 0.29.24
# pip install Cython

from setuptools import setup
from Cython.Build import cythonize

setup(
    name        = 'Reduction Test',
    ext_modules = cythonize("*.pyx"),
    zip_safe    = False,
)
```

```
#!/usr/bin/env python3
# filename: test_cython.pyx
import time

start = time.time()

cdef int i
cdef double s
cdef int[10*1000*1000] a

for i in range(10*1000*1000):
    a[i] = i
```

```
    s += float(a[i])
end = time.time()

print(f"Processing {len(a)/1000000}M elements takes: ", (end - start) * 1000.0, "
milliseconds")
print(str.format("Sum is {0:g}" , s))
```

Build and launch

```
$ python setup.py build_ext --inplace
$ python -c "import test_cython"
```

Output for Python 3.6.9:

```
Processing 10M elements takes: 19.9463367 milliseconds
Sum is 5e+13
```

---

4. C++ implementation

```
#include <iostream>
#include <chrono>
#include <iterator>

using std::cout;

namespace chrono = std::chrono;

static int a[10'000'000] = {};

int main() {
  auto const start = chrono::steady_clock::now();

  double s = 0.0;
  for (size_t i = 0; i < std::size(a); ++i) {
    a[i] = i;
    s += double(a[i]);
  }

  auto end = chrono::steady_clock::now();

  cout << "Processing " << std::size(a)/1'000'000.0 << "M "
       << "elements takes " << chrono::duration_cast<chrono::milliseconds>(end -
start).count() << " ms\n"
       << "Sum is: " << s << "\n";
```

```
    return 0;
}
```

Building Command Line for G++ 7.5.0:

```
$ g++ -O3 -DNDEBUG -Wall --std=c++17 -s test2.cpp -o testcpp
$ ./testcpp
```

Output:

```
Processing 10M elements takes 16 ms
Sum is: 5e+13
```

Comment: In Python the `float` type is equivalent to `double` in C and C++. (See sys.float_info in Python3 Library documentation).

Results. From that benchmark, we see that the C++ implementation:

- Works *x137* times faster than plain Python implementation.
- Works *x65* times faster than Python implementation that uses Numpy.
- works *x1.18* faster compare to Cython implementation.

If you need to have a highly effective algorithm implementation in Python without using translation from Python to C++ (via such a language as Cython), then it's not easy to be better than usual C++ code, even in simple things.

> Cython is a Python language with C data types. While using Cython, the source code is translated from Python into C or C++, and finally, the code is compiled as a Python extension module. As you see, compilable languages bring an extremely significant speedup. Almost any piece of Python code is also valid Cython code (See Cython limitations).
>
> Cythons has two primary use cases:
>
> 1. Extending the CPython interpreter with fast binary modules
> 2. Interfacing Python code with external C libraries.
>
> As we have observed, Cython provides a way to speed up simple single-file Python code. But usage of Cython brings you to a situation where you are not using an interpreter anymore because Cython code is compiled. Nevertheless, it can be helpful to be aware of such a possibility to improve the speed of Python code bases.

Standards for the Language

Both compiler writers and people who use the C++ language as writers should obey the international standard ISO/IEC for the language.

C++ Standardization has a long history:

- C ISO/IEC 9899:1999. Standard C99: link
- C++1998 standard - ISO/IEC 14882-1998. Draft of C++1998: link.
- C++2003 standard - ISO/IEC 14882:2003. Standard C++2003: link.
- Technical Report on C++ Library Extensions - ISO/IEC TR 19768:2007.
- C++ 2011 standard- ISO C++ standard (ISO/IEC 14882-2011). Draft of C++2011: link.
- C++ 2014 standard - ISO C++ standard (ISO/IEC 14882:2014). Draft of C++2014: link.
- C++ 2017 standard - ISO C++ standard (ISO/IEC 14882:2017). Draft of C++2017: link.
- C++ 2020 standard - ISO C++ standard (ISO/IEC 14882:2020). Draft of C++2020: link.
- C++ 2023 standard - ISO C++ standard (ISO/IEC 14882) - *Not Available*. Draft of C++2023: link.

Scott Meyers gave a presentation about C in Yandex (Russian analog of Google) back in 2014. His point of view was that C++03 can be considered a bug fix release of C++98, and C++14 completes C++11.

Language Guarantees

Fundamental code guarantees of C++ :

1. **Basic Guarantees**. This guarantee states that if an operation fails, no resources are leaked. Standard library components are designed to adhere to this guarantee, ensuring that operations do not leave the system in a bad state.

2. **Strong Guarantees**. This guarantee ensures that if an operation fails, the state of the program remains unchanged as if the operation never occurred. Therefore operation is fulfilled completely or not. All containers in C++98 provide a basic guarantee. Some operations (for example, `std::vector<T>::push_back`) give a strong guarantee.

Stages of Source Code Translation in C++

The Compiler and Linker. Briefly

The compiler converts text in a high-level language into instructions for a specific Instruction Set Architecture (ISA) of the Computing Device or another machine-dependent representation. It saves the results of processing each source file into a corresponding *compiled object file*.

Compiled object files augmented with another binary file from static libraries, are linked into the final executable. The language does not specify the internal details of the compilation or linkage - it's the responsibility of the creators of toolchains. A toolchain is a collection of system programs to organize the building process from source code to executable binaries in the target processor architecture.

The final binary format such as ELF for Linux and PE for Windows and Mach-O for macOS is also *not* under the obligation of creators of Language or user space applications developers. It's under the responsibility of the creators and authors of the authors of Operating System.

There are situations when the target device in which the program will be executed has no Operating System at all. The case of launching the program on a target device in such a case is sometimes denoted as *"Launching on Bare Metal"*. In that later case, the format of binary files is typically under the Device Vendor's responsibility (example: PTX, SASS for NVIDIA GPU is provided by NVIDIA).

The Compiler and Linker. Details

A high-level overview is presented above, but if you are curious about how a compiler compiles source code, then welcome to this section. It's possible to be productive even without the knowledge below, especially during creating only user space applications. If you want to know how things are working and you have that curiosity, you're welcome to read the text below. In another case, just skip it.

A source code for C and C++ consists of source files.

Each source file is translated (or processed) through the following sequence of steps.

Initial Textual Source Code Processing and C Preprocessing

1. The input file is read into memory, and the file is broken into lines.
2. Processing trigrams. All available C trigrams can be obtained from [2,p.15]. Processing trigrams are required for code substitution for the following code snippet:

```
#include <iostream>

int main() {
  std::cout << "Do you know C++? Are you sure ??)";
  return 0;
}
```

3. Line splicing. Joining strings through the backslash character.
4. Replace comments with white space.
5. Split program text by preprocessor tokens. The tokens for the C preprocessor are similar to tokens used by the C compiler, except for some small differences. For example, the C preprocessor token `##` is a concatenation operator for words in a source code. It is a valid operator for the C preprocessor, but it is an invalid token for the operator in a proper C or C++ program. A similar operator `#` inside the body of macros is an operator to convert the formal argument of macros into the string.
6. Processing the program code by the preprocessor. The preprocessor can be built into the compiler, or it can be an independent program. For details about available preprocessor language directives, please read [2, p.43] or documentation for any de facto standard toolchain like GCC.

The output of the preprocessing of the source file is named as *preprocessed source*, and if this stage is presented separately during compilation, the output file typically has the extension `*.i`. Obtaining such a source file can also be done separately. For clang it can be achieved via `clang -E` and similar for gcc and via

`/P` for MSVC. After the initial textual phase and C preprocessor phase, the next phase for compilation is Lexical Analysis.

Lexical Analysis

The essence of Lexical analysis of the program is in splitting the program source code text into tokens. The separate words or atoms of a program source code text are words of the source text that cannot be divided further.

One way to separate tokens is via whitespace. In C and C++, whitespace includes different forms of keyboard spaces and comments. However, sometimes tokens are not separated by whitespace (for example, using connect tokens with operators). In this case, special effort should be made.

An important aspect of C and C++ is that the C and C++ compiler always tries to assemble the longest valid token (in terms of the number of single characters) by processing the text from left to right, character by character, even if the result is an unbuildable program. Example from [2, p.20]:

```
int a = 1, b = 1, c = 3;
c = b--a;    // Compile error
// ..
// Invalid tokenization: tokens (b, --, a)
// Valid tokenization: tokens(b, -, -, a)
// ...
// But C and C++ compiler does not do that c = <b> <-> <-a>;
```

In C and C++ and in fact in most programming languages, the tokens fundamentally can be one of the following five types:

- a. Operators
- b. Separators
- c. Identifiers
- d. Keywords
- e. Literal constants

Employed algorithms at this stage include deterministic/non-deterministic Finite Automata and Substring Search. After finishing this phase, named the Lexical Analysis, the program consists of a sequence of tokens.

Syntax Analysis

The compiler's understanding of the program is partially based on the language rules typically described by Backus – Naur forms for Context-Free-Grammars (CFG) which describes which valid lexemas/tokens constitute a valid program. The Grammars by themselves are studied in a mathematical area called *Formal Languages and Grammars Theory*. That area of mathematics is essential for some Compiler's fundamental aspects.

Based on the grammar of the C or C++ programming language, the syntax analyzer constructs the Abstract Syntax Tree (AST) for the program's source text. The exact grammar rules can be found in the Appendices of the corresponding Language Standards.

Employed algorithms for constructing the parsed representations depend on implementation, but they can be, for example:

- A Recursive Descent Algorithm without Backtracking
- Recursive Bescent with backtracking
- Predictive Top-Down Parsing
- Parsing Tables for LL(1) and LL(k) grammars
- Bottom-up parsing with shift-reduce parsing

Semantic Analysis

Unfortunately, some rules of the language can not be expressed only by using CFG. Examples: multiple declarations of a variable in one scope, usage of not yet declared variables, access to a plain C array via an index that is out of range, etc.

For handling such analysis, which can not be captured by a syntax analyzer, the semantic analyzer inside the compiler is used. In some sense, not everything can be captured with strict grammar rules.

Code Optimization

At this moment, we constructed an Abstract Syntax Tree(AST) for a program and augmented it with information from the semantic analysis stage.

Also, we can traverse AST and translate this code into a more low-level construction expressed as Assembly Language (ASM) or Intermediate Representation (IR).

Compilers' innovations based on various fields of science and engineering that mainly bring considerable speedup are exploited in this stage!

Typically, compilers perform a sequence of transformation passes. Each transformation pass analyzes and edits the code to optimize performance. A transformation pass might run multiple times. Steps run in a predetermined order that usually seems to work well. Some examples of optimization techniques that are happening at this moment and considered as a general practice:

- Convert one arithmetic operation into cheaper operations by using bit tricks and logic/arithmetic shifts.
- Replace stack allocation storage with storing variables in the processor's register.
- Optimization of structure/class memory layout.
- Transform data structures to have the ability to store elements of it as much as possible in CPU registers when some function obtains input in the form of a pointer/reference of an object of the structure/class type.
- Remove dead-end code that never executed in the program's control flow across compiled source files.
- Function inlining. The compiler uses its heuristics to decide what to inline and what not to in the program code. Sometimes there is a toolchain extension that forces the compiler to make a specific decision or provides means to tune heuristics slightly.
- In the case of using a global program optimization compiler and linker jointly, you may want to try inline even function definition from another compilation unit.
- Try to remove the recomputation of the loop-invariant code inside the loops. The technical term for this is Hoisting (or loop-invariant code).
- Vectorization. Leverage into vector registers (like SSE2, Arm Neon, etc.) when possible.

- Loops Unrolling. Unroll several-iteration of the loop to reduce control overhead and sometimes open the ability to leverage Instruction Level Parallelism with the price of code size.
- Loops Fusion. Loop Fusion, also known as *jamming* combines multiple loops over the same index range.
- Figure out with Memory Aliasing and apply optimization for non-aliased pointed expression (For details, please check Compute Optimization Relative Information).
- Constant Propagation. A big part of the compiler and benefit of having a compiler is that it can propagate constants across the whole source code of the program. This is utilized to derive values during compile time.
- Register allocation. To decide in which moment of runtime different registers can be used to store values the special algorithms/policies should be used.

Various things of optimization are out of the scope of the compiler. And can only be solved by the creator of the Algorithm.

> We think that is a nice research direction when such algorithmic aspects are provided for the compiler.

Code Emitting

In the end, at least conceptually, the compiler emits final instructions for the target Assembler. How exactly emit code is under the decision of the compiler and toolchain creators.

However, in reality, it's possible to have three different scenarios of what exactly compilers emit:

1. The compiler emits the final binary code for the target Instruction Set Architecture (ISA). (Example: Microsoft Visual C compiler does that.)

2. The compiler emits the program text written in Assembly. But in fact, the process of producing the final binary code is under the responsibility of the Assembler Program. You can obtain such assembly source from a preprocessed file manually for clang toolchain via invocation of `clang <source_file.i> -S -o -`.

3. With the coming LLVM project, there is an intermediate layer between High-Level Language (such as C++) and ASM for the target device (described by ISA). This layer is called Intermediate Representation (IR). It contains three stages of conversion:

    - At the *first stage* the input preprocessed code is converted into a pseudo-assembly called LLVM-IR producing files with extensions `*.ll`. You can obtain unoptimized LLVM-IR code in clang toolchain via invocation of `clang <source_file.i> -S emit-llvm -o -`.

    - At the *second stage* LLVM-Optimizer works under such representation and produces optimized `*.ll` source code.

    - At the *third stage* the **LLVM code generator** generates real Assembly representation.

For further study as an introduction to LLVM-IR, we recommend Lecture 5 from the MIT course 6.172 Performance Engineering of Software Systems.

Calling Assembler Program

Assembler(ASM) Language is the lowest possible level that can still be readable, but understanding it (without extra tools and extra documentation) is not easy in big programs. ASM language has a close relation to target computing devices.

One instruction in C++ code can correspond to several(1,2,3, etc). ASM code instructions. On the other hand, the same instruction in C and C++ can be emitted (materialized or generated) into different instructions in the ASM Language.

An Assembler is a program that finally converts ASM instructions obtained from a compiler into binary native code for the target device. The machine instruction emitted by ASM is described by the target Instruction Set Architecture (ISA). In Assembly literature, the process of converting ASM instructions into machine code is named `encoding`. An inverse process of reconstructing ASM code from binary code is called `decoding` or `disassembly`.

For GCC toolchain, the standard de-facto Assembler is GAS. The output of the Assembler is saved into *object files*.

The Assembly output code by itself obeys Instruction Set Architecture *ISA*. The *ISA* specifies instructions, registers, memory architecture, data types, and control flow mechanisms. The ISA connects physical Hardware designed by Electrical Engineering (EE) with the software constructed by Computer Science (CS).

There are online tools such as [11] Compiler Explorer that provide a demonstration of the generated Assembly code while using various compilers and target platforms for C++ online. It can be very useful for educational and analytical purposes because using color shows the correspondence between C++ code and Assembly code.

The particular implementation of Instruction Set Architecture (ISA) is called Microarchitecture in Electrical Engineering (EE) terminology. Different vendors can provide the support of the same ISA, but Microarchitecture is typically under a Non-disclosure agreement (NDA). Microarchitecture is the lowest level of computation and it's under the responsibility of Electrical Engineers.

Linkage

The linker constructs the final program or dynamic (shared) library from compiled source files in the form of *object files*, obtains additional input archives of object files (called static libraries), obtains information about used dynamic library dependencies, performs other semantic checks (for example via finding undefined references for C and C++ entities), using specially provided flags, perform a whole-program/global program optimization or optimization specified via command line for it.

The nuances of compiler/linker organization are out of the scope of the C++ language and can vary from vendor to vendor. For example, in GCC, the Assembler is a separate program. Its execution is carried out independently from the C compiler. In other toolchains, e.g., Microsoft Visual C compiler, the translation to the final binary code is located inside their C compiler.

The name of the linkage program typically in toolchains has a name such as ld or link.

What is Impossible Even in C and in C++

- **Address individual bits.** Few machines can directly address an individual bit on the level of the implemented ISA. Even if some ASM allows it, it is out of the scope of C and C++, to be honest. Of course, you can operate on bits, but not directly.

  > For whom it may be interesting - conceptual accessing individual electrical wires (or bits) is available via Hardware Description Languages (HDL) which are used to describe electrical circuits to build electrical computing and storage components. However, this level is far below Assembly (ASM) and out of the scope of this technical note. But such level is on the boundary between Computer Science and Electrical Engineering exist via exploiting Hardware Description Languages with reconfigurable compute devices.

- **Define your operators syntactically with their syntax.** B.Stroustroup, in his Technique FAQ ([8]), said that the possibility had been considered several times, but each time they decided that the likely problems outweighed the potential benefits.

For People New to C++

If you have arrived from another programming language which C-like, but you are only a bit familiar with C++, we would like first to enumerate several things that are not complicated for you.

1. The logical operators for logical "and" `&&` and logical "or" `||` are called short-circuit logical operations. This is due to their behavior of evaluating subexpressions in the chain of logical operations only when needed for the logic expression evaluation. The bitwise operators `&` and `|` do not perform short-circuit.

2. The difference between the two pointers at the level of the C and C++ languages is measured in terms of elements, not in terms of bytes.

3. In nested statements, an `else` always belongs to the nearest preceding `if` as in most programming languages. The potential confusion in the context of programming languages is known as the *dangling else problem*.

4. Every heap memory allocation `new` must be paired with a single `delete`. Every `new[]` must be paired with a single `delete[]`. Any other sequence of events leads to either undefined behavior or memory leaks.

5. In the standard library API for different forms of strings API, the convention is that you provide *first index* and *length*. Unfortunately, in some languages (Java, Python), the design is different - you provide *first* and *end* indices. Our biased opinion is that the C++ choice is better because it can be the case that if the length is compile-time constant, and the compiler can optimize the code better.

6. You should never return the address of an automatic, stack-allocated local variable from a function. Stack-allocated objects (instances of the class and structs) or built-in variables allocated in the stack are allocated in the stack. The stack is cleaned after function execution.

7. When a non-static class member function executes, it conceptually automatically contains a hidden pointer with the name `this`, which includes the address of the object for which the function was called.

8. You can only call `const` member functions for `const` objects. You should, therefore, specify all member functions that do not change the object for which they are called `const`.

9. Static member functions cannot be `const`, because a static member function is not associated with any class object (or with any other object at runtime). Therefore, there is no `this` pointer, and the `const` property is not applied to them. Making `const` static methods of the class leads to a compile-time error.

10. The primary purpose of operator overloading is to increase the readability of the code.

11. The notation for calling the base class constructor is the same as that used for initializing member variables in a constructor.

12. Every derived class constructor is called a base class constructor. If **a user-defined class constructor** does not explicitly call a base constructor in its initialization list, the default constructor will be called implicitly. Example:

```
#include <iostream>

class A {
public:
  A(){std::cout << "A()\n";}
  A(const A&){std::cout << "A(const A&) not called\n";}
};

class B: public A {
public:
  B(){std::cout << "B()\n";}
  B(const B&){std::cout << "B(const B&)\n";}
};

int main()
{
    B b1;
    B b2(b1);
    return 0;
}
```

13. A table of virtual function pointers is created for each class and contains a table of virtual function pointers. The general advice is that the only time you should even debate whether the overhead of a virtual function is worthwhile to optimize it is when you have to manage many objects of the corresponding type.

14. The general form of a pointer to a function definition is as follows:

    `return_type (*function_pointer_name)(parameter_types);` [4, p.733].

15. There are names reserved for the Compiler Implementations. It's all names that:

    - Starting with a double underscore `__`
    - Names that start with a single underscore `_` followed by an uppercase letter `_[A-Z]`, e.g., **_Foo**.

That name should not be used by non-compiler and non-STL writers, both for the C and C++ languages. With that rule for a long time, people have escaped conflicts between the naming of compiler-specific entities and entities of the constructing program.

> Sometimes compiler writers violate this principle. For example, The Linux Programmer's Guide mentions that standard predefined macro uses `unix` and `linux` during compilation for the Linux platform. These special names are *not* the names that follow C and C++ conventions.

> Sometimes, it's possible to observe defined guards at the beginning of the buildable program's header files via `#ifndef/#define` that uses names starting with `__`. These special names are not the names that followed the mentioned rule and these names are incorrect usages of identifiers. In modern code (from C++11), it's less likely to find define guards because the pretty universal approach is to use compiler extension `#pragma once` for this purpose, which is universally supported.

16. In C and C++, postfix operators have a higher priority than unary operators.

    ```
    *p++; // *(p++)
    ```

17. In C and C++, unary operators have a higher priority than binary operators.

18. Operators `==`, `!=` have higher priority than logical connectives. See also: https://en.cppreference.com/w/cpp/language/operator_precedence

19. The C++ standard guarantees that the life of a temporary object if it is **LValue** (the object that occupies memory) is extended to the life of any reference that refers to it, **including the constant**. In simple cases, based on this trick, you can capture returned temporary objects from a function by constant reference to reduce the number of copy constructors. To utilize this trick, the function must still return an object by value. Returning a temporary object by reference is incorrect, and it will lead to the already-mentioned problem (6).

20. References to **RValue** objects whose address cannot be obtained do not extend the lifetime of temporary objects. However, the compiler can only detect simple constructions with **RValue** objects. For example, if you create a temporary object and call a method from this temporary object, that will return a reference to itself. The compiler will no longer be able to determine that this reference is not valid after removing the temporary object. Therefore, you should be very careful with this case (20) and the previous case (19).

21. The call for containers in standard C++ with name `empty()` does not empty container. It returns the answer to the question "*Is the container empty?*".

About C and C++ Preprocessor

C++ 1998 and C++2003 use the C89 preprocessor, although the C language also has evolved: Traditional C, C89, C95, C99, C11, and C17. For a detailed description of the C preprocessor, please read Chapter 3 in ([2]). Below, we provide a short overview of it.

The C preprocessor commands start with the `#` symbol in the source code. The C language allows any number of whitespaces before and after the `#` symbol. If the line contains only `#`, this is an empty preprocessor command, and it is ignored. This is what is called a "null directive" in C preprocessor language. Next, the C preprocessor can generate, in principle, an invalid C program.

In the function-like macro definition as in `#define sum(x,y) ((x)+(y))`, there should not be a whitespace between "m" and "(". Function-like macros may have zero parameters.

There is no requirement that all arguments should be used in the body of the macros. Preprocessor macro extensions in C and C++ have the following important properties. Once an extension replaces a macro call, the macro call search process starts from the beginning of the expanded extension to find further replacements.

However, during this process, macros referenced in their expansion are not re-expanded, and that preprocessor macro extension does not lead to infinite recursion. Example:

```
#define sqrt(x) (x<0 ? sqrt(x) : sqrt(-x))
```

If C macro expands and generates text which by itself will be a macro definition commands, this last macro definition commands will not be treated as a preprocessor macro definition (but it can contain calling of already defined macros, so be careful).

For the C and C++ preprocessor, any undefined identifiers (i.e., identifiers that do not correspond to the definition of macro) that appear in expressions after the conditional directives `#if` and `#elif` are replaced with the number `0`.

Include Search Order

The original C specification says that the actual directory in which the compiled source file is located is used to look for a user-defined *include file*.

But nowadays, the order of include paths varies between compiler toolchains, so you may figure it out for a particular toolchain by experimenting with it or from its documentation.

Include Files Naming

For each "C" standard library `<X.h>` header file, there is a corresponding C++ standard header file `<cX>` in C++. A standard header file whose name begins with the letter `c` is equivalent to a standard header file in the C library. (B. Stroustrup, Spec. Edition, p. 487, 16.1.2). Those header files expose different behaviors in terms of using their names from the global namespace:

- Standard headers with naming as `<X.h>` define function names in the `std` namespace and also **import those names into the global namespace**. So `<X.h>` means "a C style" including compatibility.
- Standard headers with naming as `<cX>` define function names **only** in the `std` namespace. ([1], 9.2.2, page 247).

Predefined Identifiers and Macros

Macros	Meaning
`__func__`	In C99, a predefined identifier with the name of the current function. C++11 officially supports that too.
`__LINE__`, `__FILE__`	Current line number, and current source file name.
`__DATE__`, `__TIME__`	Date and time of source file compilation.
`__STDC__`	Compiler conforms to the C standard.
`__VA_ARGS__`	Only C++11 and C99 formally support that, but informally `__VA_ARGS__` is often supported. This built-in name can be used for macros with an arbitrary number of arguments. When the macro is invoked, all the tokens in its argument list `...`, including any commas, become the variable argument.
`__cplusplus`	Version of C++ standard that is being used. For MSVC, you should provide a compiler option `/Zc:__cplusplus` to ask the toolchain to define this variable properly.
`__STDC_VERERSION__`	Version of standard C.

Example with using `__VA_ARGS__`:

```
#define my_printf(...) \
do{ fprintf(stdout, __VA_ARGS__); } while(0)
```

Example of checking the version of C or C++ compiler based on [2, p.53]:

```
#include <stdio.h>

int main() {
#ifdef __cplusplus
  printf("C++ version %li\n", __cplusplus);

#elif defined(__STDC__)
#   if defined(__STDC_VERERSION__) && __STDC_VERERSION__ > 199901L
  printf("C99 standard\n");

#   elif defined(__STDC_VERERSION__) && __STDC_VERERSION__ > 199409L
  printf("C89 with additions 1\n");

#   else
  printf("C89\n");

#   endif
#else
  printf("C not standartizied\n");
#endif
```

```
    return 0;
}
```

Language Rules

Names Overloading

In C and C++ and potentially in other programming languages, the same identifier can be associated with more than one object at a given moment. This situation is called *name overloading* or *name hiding* ([6], chapter 13).

Creating two declarations of the same name in the same overload class in the same visibility block or at the top level is an error.

#	Overloading class name	Identifiers included in the class
1	Preprocessor Macro Names	The names used by the preprocessor are independent of any other identifiers.
2	Operator labels/tags	The labels used immediately follow the `goto` statement.
3	Structures, Union, and Enum tags	They are part of a structure, union, or enumeration and immediately follow the keywords: `struct`, `union`, and `enum`.
4	Components namespace	Defined in the namespace(or name scope) associated with the corresponding structure or union type.
5	Another namespace	Name of the following objects: *Variables*, *Functions*, *Typedef names*, *Enumeration constants*.

C++ introduces structure and union tags, and enumeration names are implicitly declared via `typedef` in the namespace "*Another*" where there are also usual variables.

If you explicitly use a `typedef` for a structure followed by a variable declaration, it will lead to an error.

However, tag names can be hidden by subsequent variable or function declarations or by an enumeration member of the same name in the same scope.

Interestingly, according to ([6], Section 3.3.7), functions/variables take precedence over type tags in any order.

Literal Constants

In C++, in a literal expression, you can encode the type of literal:

#	Type	Suffix	Alternative suffix
1	`long`	`l`	`L`
2	`long long`	`ll`	`LL`

#	Type	Suffix	Alternative suffix
3	`unsigned`	`u`	`U`
4	`unsigned long`	`ull`	`ULL`
5	`float`	`f`	`F`
6	`double`	no suffix	
7	`long double`	`l`	`L`
8	`std::string`	`s`	
9	`std::size_t`	`z`,`uz`	`Z`, `UZ`

Suffixes (1-7) have been available since C89.

The suffix (8) `s` has been made available only since C++14 (link). The suffix (9) `Z` has been made available only since C++23 (link).

Prefixes for Strings from C++11

Starting from C++11, you can use the following prefixes for strings:

#	Suffix	Description
1	`L'a'`	`wchar_t` symbol. Typically for Windows it's UTF-16, for Linux it's UTF-32. Available from C++98.
2	`u'a'`	UCS2 symbol. Pretty like UTF-16, but surrogate pairs are not supported in UCS2.
3	`u"a"`	UTF-16 string. With support for surrogate pairs.
4	`U'a'`	UCS4 (UTF-32) symbol.
5	`U"a"`	UCS4 (UTF-32) string
6	`u8"UTF8_string"`	UTF-8 string
7	`R"(asd\n)"`	Raw string. Analogue of Python's `r"""str str"""`. Multiline is supported for such lines, and special character sequences \n are not special.
8	`R"*(asd\n)*"`	Raw string literal with custom delimiters.

The UTF-8 and UTF-16 are variable-width encodings for characters.

Function Call Nuances

1. There are only two kinds of functions and function calls in C++:
    - *Ordinary Function Call.* This is a call to a function. A static member function is an ordinary function ([6], 9.4).
    - *Member Function Call*. This is a call to a member class function, also known as a class method.

2. In functions with `void` return types or when the return type is absent (e.g., in constructors/destructors), you can have the absence of a `return` statement in a function body. It is equivalent to an explicit `return;` at the end of the function body.

3. Due to ([6], 6.6.3) "*Flowing off the end of a function is equivalent to a return with no value;* ***but this results in undefined behavior in a value-returning function***."

4. In C++, the `main` function cannot be called recursively.

5. Starting from C++11, there is a suffix syntax for the function return type. It is not primarily about templates and type deduction - it is about scope. One more example of when it was useful C++11 Far from B.Stroustroup ([9]).

   ```
   template<class T, class U>
   auto mul(T x, U y) -> decltype(x*y) {
     return x*y;
   }
   ```

   The notation of auto means "*return type to be deduced or specified later*".

6. A non-constant reference cannot refer to a temporary variable.

7. Although temporary objects can only be passed as `const T&` or `T`. However, calling non-const methods on temporary objects is allowed. The initializer for `const T&` does not need to be an LValue and even be of type `T`. In such cases, a temporary variable is created to hold the initializer, lasting until the end of the scope of the reference.

8. Linkage rules cover name mangling and the call convention. Due to ([6], 7.5.3), there is the following requirement for the linkage aspect of functions:

   Every implementation shall provide for linkage to functions written in the C programming language "*C*" *and linkage to C++ functions* "*C++*". Example:

   ```
   extern "C++" void f(); // To link functions in C++ style:
   extern "C" void f(); // To link functions in C style:
   ```

Requirements for C++ Expressions

1. In C++98/03, you cannot modify a variable more than once without a sequence point in C++03/98. *Sequence point* - semicolon, function return, function jump, and some others.

2. C++11 introduces the term *order of evaluation*. The term *sequence point* is no longer used. But the rule is the same.

3. In C++17, the statement in which you modify the variable more than once is a bad practice. However, the C++17 standard added the rule that **all side-effects of the right side of an assignment** are fully committed before evaluating the left side and the assignment operator.

That means that the following statement is a legal expression starting from C++17:

```
int k = 0;
k = k++ + 5; // Valid from C++17
```

Exceptions to the One Definition Rule

You can read about that in detail in ([1], 9.2.3 p. 248). But essentially, it can be more than one definition of the following things in different translation units (C++ source files):

1. Class type
2. Enumeration type
3. Inline function definition with external linkage
4. Class template
5. Function template
6. Static data member of a class template
7. Member function of a class template
8. Template partial specialization (template for which some template parameters are not specified)

The same definitions are acceptable when:

1. They are located in different translation units
2. Their definitions are identical, token by token
3. The meaning of token is the same in both translation units (Checking this is not included in the capabilities of the programming language itself and it is assigned to the building toolkit.)

Details about exceptions to the One Definition Rule are described in ([6], C++2003, p.23, §3.2/5).

Integer Arithmetic and Enumerations

In C and C++98/03/11/17, there are three allowable implementations for signed integers consisting of n bits. For details, we recommend looking at [2, p.125], but some information about them is presented below:

- *Two's complement (or twos-complement-notation)*. The range is: $$[-2^{n-1}, 2^{n-1}-1].$$

  Positive numbers are represented in the usual way. The most significant bit of the sign is set to 0. Negative numbers are obtained via reversing(flipping) all bits of the positive number representation plus 1. In Assembly for x86 this can be achieved via using NEG operation.

  `1000 ... 0000 0000 (bin)` is the maximum negative number that has no positive equivalent.

- *One's complement (or ones-complement-notation)*. The range is: $$[-2^{n-1}+1, 2^{n-1}-1].$$

  Negative numbers are the complement of all bits of the corresponding positive number. In this representation, positive and negative zeros are possible. The implication is that representation has one number less than *Two's complement*.

- *Signed integer representation (or sign-magnitude-notation)*. The range is: $$[-2^{n-1}+1, +2^{n-1}-1].$$

The representation of the modulus of negative and positive numbers is identical bit to bit. But the sign of the number is stored in the most significant bit.

It turns out that the most popular representation for signed integers by hardware vendors is the *two's complement* notation.

> Starting from `C++20`, there is only one signed integer representation in the language and its twos-complement notation.

A particular dedicated type for enumerating integer constants from C89/99/11 and C++98/03/11 is called `enum`. There are some subtleties with it:

- In C98/C99, the `enum` type is a synonym for integer with type `int`.
- Unlike C98/C99, the C++ language treats each enumerated type as a specific type and from integers as well.

In reality, the underlying type for C++ for implementing `enum` has some underlying integer type to store values, but it is not specified by the language.

However, starting from C++11, we now have two types of `enum`:

Strongly Typed Enums(or scoped enumerations).

Example:

```
enum class Color : int{red,green,blue};
```

Strongly typed `enum`:

- Does not support implicit conversion to `int`.
- For them, it's possible to specify the underlying type.
- The names of enumerated elements are defined *only* in a namespace of the enumeration type.
- The default underlying type for a strongly typed `enum` is `int`.

Usual enums (or unscoped enumerations).

Example:

```
enum Color : int{red,green,blue};
```

For ordinary `enum` starting from C++11:

- It's possible to specify explicitly the underlying type
- Nothing is said about the underlying type by default in C++11, as in C++98
- B.Stroustrup, in his blog, notes that it is now possible to explicitly (optionally) use the name of a regular and scoped `enum` as a namespace to refer to its elements
- For the ordinary `enum` mechanism for accessing `enum` elements through enclosed namespaces is still in the language, even though sometimes it is not what you want.

- If you specify the underlying type (and so at least the size is known for the compiler), you can make a forward declaration for `enum class` and `enum`.

Example with the forward declaration:

```
//enum A; // Compile Time Error: Forward declaration for unscoped enums is not
allowed

enum  A {
    a = 1,      // [X]
    b= 2
};

enum B:std::uint64_t;

enum B:std::uint64_t {
    c = 1,
    d= 2,
    // a = 3 // Compile Time Error Due to [X]
};
```

Documentation: cpp reference enum

Integer Types Nuances

1. The compiler's use of `unsigned` or `signed` in the absence of an explicit type specification is not defined for the `char` type.
2. The compiler's use of `unsigned` or `signed` in the absence of an explicit type specification is not defined for the **bit fields**
3. In general, the result of doing the right shift for signed integer types in the case of negative numbers was undefined before C++20. The compiler implementer can perform either a *Logical Right Shift* or *Arithmetic Right Shift* ([2], p. 252).

> From C++20, the right shift on signed integral types is an **Arithmetic right shift**, which performs sign-extension.

Auto Type Deduction

The `auto` in C++98/03 and in C89/99 was an explicit (and default) memory type modifier for local objects located in the stack (link). Starting from C++11, the keyword obtained another meaning - it's a type automatically deduced, similar to `template` type arguments in template functions.

Also, `auto` never deduces to a reference type, always to a value type.

This implies that the value still gets copied even when you assign a reference to `auto`. To make the compiler deduce a reference type, you can use [4, p.309]:

- `auto&`

- `const auto&`

The `auto` works in the usual functions for creating variables:

```
auto v1(expr); // Direct initialization
auto v2=expr;  // Copy initialization
```

And create variables inside range-based loops:

```
int arr[] = {1,2,3};
for (auto x:arr) {
  printf("%i\n", x);
}
```

You need to be careful when using braced initializers with the `auto` keyword because it will deduce to std::initializer_list:

```
auto x1 = {1,2,3,4}; // x1 is an initializer_list<int>

// Bracket/Braced/Uniform initialization via using {}
// In the form of list initialization does not allow narrowing
```

The `auto` can be used jointly with constant volatile type qualifiers (cv) and references and pointers. Example:

```
int ii= 1;
const auto* p_ii = &ii;
const auto& p_ref = ii;
const volatile auto* p_iii = &ii;
```

The *volatile* is a type qualifier that denotes that the type can alter its value not due to the C++ language but for other system reasons. Thus, the C++ implementation should be careful with optimization access for that variable through registers.

> The *volatile* variable will not be located in the register (the compiler should guarantee this), and read/write will not be optimized and will go through memory (at the assembly level). In C++, the usage of `volatile` does not imply a `memory fence`, so be very careful with creating multithreaded code with it. The memory fence (or memory barrier) means that memory operations started to be issued before the barrier are guaranteed to be performed and completed before operations after the barrier. Once memory is recorded in the Cache, the Cache Coherence and Consistency protocols will guarantee correct operations at the hardware level. Saying all that, you still should use memory fences appropriately when you need to reconstruct sequence memory consistency guarantees in multithreaded applications.

The close-by conception for auto is decltype. It provides the ability to derive the type of expression without evaluating the expression at runtime.

> There are some subtleties with `decltype`. The `decltype(x)` and `decltype((x))` are often different types. If the argument for `decltype` is an unparenthesized expression `decltype(x)` or unparenthesized class member access expression, then decltype yields the type of the entity named by this expression. The inner parentheses `decltype((x))` cause the statement to be evaluated as a type of an expression itself, and it typically will be a reference.

Example:

```
int main()
{
    double z = 1.0;
    struct Point
    {
        double x;
        double y;
    };
    const Point a = {10.0, 11.0};


    decltype(a.x) x4;     // type is double; decltype(expression) is the type of
the expression

    decltype((a.x)) x5=z; // type is const double&

    x4 = 123.0;
    // x5 = 567.0; // compile-time error

    int x;
    const int *ptr = &x;
    decltype(x) x1 = 1;       // x1: int
    decltype(ptr) p1 = 0;     // p1: const int*
    //decltype((ptr)) p2 = 0; // p2: const int*& compile-time error
    decltype((ptr)) p3 = ptr; // p3: const int*&

    return 0;
}
```

Extra parentheses are used to preserve the `const` property for the type of expression.

Documentation: cpp reference decltype

Range-Based For Loop

Starting from C++11, there is a new syntax for the `for` loop named "*range-based for loop*". Example:

```
for (auto i: v)
  std::cout << i;
```

Range-based loops are valid for any type supporting the notion of a range. To support range-based for loops, one of the following should be valid:

- `obj.begin()` and `obj.end()` are valid C++ expressions
- `begin(obj)` and `end(obj)` are valid C++ expressions
- Your container is the built-in array

Technical Differences between C and C++

Many times in the past, people stated that C++ and C are different. Let's take a concrete look at what it means. All the differences are pretty subtle, but there are plenty of them. The text below covers the difference between C99 and classical C++03.

---

1. Old C-style function declarations are not allowed in C++

```
/* Obsolete function definition for C++ */
double alt_style( a , real )
    double *real;
    int a;
{
    return (*real + a);
}
```

2. C programs should not use names that are keywords in C++ if one wants to be compatible with portability to C++.

3. C++ style comments `//` only appeared in C99.

4. C++03 has new operations `".*", "->*", "::"` which are not available in C.

5. Different memory for char literal in C and in C++

```
sizeof('a') == sizeof(char) // C++
sizoef('a') == sizeof(int)  // C
```

6. C++ 1998 uses the C89 preprocessor, although the C language has changed: Traditional C, C89, C95, C99, C11, C17.

7. Struct tags in C++ are included in the "other names" namespace. In this space are:

- Variables
- Functions
- Typedef names

- Enum constants

  Therefore, `struct n{}; typedef double n;` is correct in C but not in C++. However, there is one exception, which is described next.

7. Although for C++ type tag names (for `struct`, `union`, `enum`) are implicitly declared using `typedef`, they can still be hidden by variables in the same scope `S S;`.

8. C99 has the pointer qualifier `restrict`, which is not in the official specification of C++98/03/ and is supported by C++ compilers typically as `__restrict` or `__restrict__` extension universally.

9. Support of **flexible array type** array. In C99, such a concept is defined in (ISO/IEEC 9899 C99, 6.7.2.1). Essentially, it's the situation when the last element of a structure has an incomplete array type.

   ```
   struct s {
     int n;
     double d[];
   };
   ```

   In the case of evaluating the size of structure `s` - the size of the structure's element *d* is omitted in the returned value. However, it's possible to access elements of array `d` through the pointer or reference to structure `s`. In that case, you should understand what you're doing - the structure has a memory layout that maps it into the underlying buffer correctly.

10. In C99 (**but not in C++**), there is support for a `variable-length array` (defined in ISO/IEEC 9899 C99, 6.7.2.1) that arrays with a size specified via a non-const variable. Such a concept allows a portable way to perform varying allocations of automatic variables in the stack.

11. Different initialization of the char array. In C++, the array must be of sufficient size to hold the "\0" character

    ```
    // OK in C, OK in C++
    char a[3]="12"; char aEquiv[]="12";

    // OK in C, Not OK in C++
    char a[3]="123";
    ```

12. C99 has named initializers for structures and positional initializers for arrays. C++03 does not have them. In C++20, the **named initializers** from C99 came with the name designated initializers.

13. The definitions of `struct` and `union` in C++ have block scope.

14. Type of memory.

- The `const` variable declarations:
  - are `static` by default in C++
  - are `extern` by default in C

- The `non-const` variables declared on the namespace level have extern linkage by default in C++. (Appendix C. C++2003) 7.11.6, C++2003:

  "A name declared in a namespace scope without a storage-class-specifier has external linkage unless it has internal linkage because of a previous declaration and provided it is not declared const. Objects declared `const` and not explicitly declared extern have internal linkage."

15. C++ declaration `void f()` is equivalent to void `f(void)` in C. The declaration in C `void f()` states that the function has an indefinite number of arguments.

16. If the array is multidimensional, then in all cases, only the leftmost index can be omitted to determine the array's size. In this case, the array size will be derived from the initializer size. Also, in C99, component-wise initialization is allowed, which is known as designated initializers in C++20. What is known as position initialization in C99 is not supported in C++:

    ```
    int a[4] = {[1]=2};
    ```

17. In C, but not in C++, you can write, although this is strange: `sizeof(struct S{int a;});`

18. Implicit cast from integer type to `enum` is allowed in C but not in C++. ([6], p. 113. 7.2.5):

    "The type of an `enum` is an integer type that must support all underlying values. In C, enum has a synonym for int."

19. In C++, converting a *void pointer* to any reference type requires an explicit cast operation. In C, this is done implicitly.

20. Unconditional branching through `goto` is allowed in the middle of a nested block in C (while initialization of automatic variables is not guaranteed). This is not allowed in C++ in general.

    > But there is an exception. The exception is for POD types - you may skip their initialization. However, there are no guarantees for the initialization of variables with POD Type.

21. In C++, there are more stringent requirements for inline functions - an `inline` function must be declared similarly in all source files. In C99, this is not the case.

22. In C++, an `inline` function, in terms of code, can have an address and static variables inside. In C, it is not allowed.

23. In C99 and C++, the compiler must see the function definition, i.e., the function should be defined so that it is `inline` in `*.h`. The compiler can choose what to do:

    - Inline all calls to the function
    - Do not inline any calls to a function
    - Partially inline, i.e., inline some function calls

24. C++ allows declaration in conditions, and it's not allowed in C.

25. There is no such thing as a reference in the C language. References are described in "C++2003, 8.3.2. References".

26. There are no overloaded functions in C.
27. C++ has default parameters, and they are not allowed in C.
28. In C++, there is a namespace mechanism, which does not exist in C.
29. In C, you can use `exit()` and `abort()` with no problems, but in C++, the destructors of local objects will not be called in this case.
30. Overloading of operators and functions is allowed only in C++.
31. C does not support general-purpose programming with templates.
32. C99 has a predefined identifier `__func__`. This identifier is implicitly defined by the compiler at the beginning of the function body as `static const char __func__[] = "function-name"`. Such an identifier was absent in C++98/03 but has been available since C++11.
33. In C++, operators not presented in C language usually have the highest precedence priority, except for `throw` which is only above the comma operator `,`.
34. In C, there must be at least one element in the initialization list when a structure or array is initialized. For C++, if an empty initialization parenthesis for a built-in array is used, the array will be zero-initialized zero initialized.
35. Before C99 (i.e., in `C89`, and in `C89 with the extensions`), there was a restriction on where automatic (stack) variables can be defined - only at the beginning of a local block. In C++98, you can declare local variables anywhere in the local scope.

Memory

Memory Types and Pointers

1. An ordinary string literal has the type "*array of n const char*" and static storage duration.
2. About a pointer to constant data and a constant pointer, it's possible to read from ([2], p. 105). In principle, a possible trick to remember this is to read the expression from right to left, and to which "*const*" type or variable is closer to that and apply this modifier.

```
int* const const_pointer;
const int* pointer_to_const;
```

3. Quite a lot of important information is contained in ([6], 5.3.3 `sizeof`), including the following:
    - `sizeof(char)` - with all variations of char is always *one byte*.
    - `sizeof(bool)` - is implementation-defined.
    - `sizeof(wchar_t)` - is implementation-defined.
4. Also, `sizeof` of structures in C and in C++ is equal to the amount of memory to store all components, space for padding between components, and space for padding after structures.

5. The `sizeof` operator is applied to an array of structures and to other types. The following rule must be fulfilled: *The size of an array of N elements in bytes is equal to N times the size of the array element.*

6. In C and in C++, a function pointer expression can be used to call a function without explicitly dereferencing the pointer, i.e., you can call the function by using the function pointer `f` via `(*f)()` or via `f()`.

Used Memory for Types and Their Layout

1. The representation of an object in memory is a sequence of bits. The representation does not have to use all the *bits*, but the size of an object is the number of *memory units* of memory it occupies.

2. The amount occupied by one `char` character is taken as a memory unit. The number of bits in a character is specified in the `CHAR_BIT` macro in the C Language. All objects of the same type, according to C and C++ rules, occupy the same amount of memory. In practice, however, one char is "always" one byte, i.e., the *8-bit* number.

3. Computers are classified into two categories in order of bytes in a word:

    - *Right to left* or *Little - Endian* - the address of a 32-bit word matches the address of its least significant byte (Examples of CPU architectures are Intel x86, Pentium)
    - *From left to right*, or *Big - Endian* - the address of a 32-bit word matches the address of its high-order byte (Motorola).

    Some systems support two modes at the same time.

4. In some computers, data can be located in memory at any address; in others, alignment conditions are imposed on certain types.

5. A typical data type to store the address of some object/data is a pointer. To store (or serialize the value of the pointer) in some integer variable, you can use `uintptr_t`. The `uintptr_t` integer type was introduced in C99. The `uintptr_t` is sufficient to store a pointer to any data, but not formally to a function.

6. A special value in C and C++ called a null pointer is equal to a null pointer constant. A null pointer can be converted to any other type of pointer.

7. A null pointer in C and C++ is:

    - An integer expression that yields zero.
    - Or an integer expression cast into a pointer.

    The expressions below will not result in a compilation error, as much as we would like them to:

```
        void y(int*){}
        y(0);
```

In C++11, in addition to NULL, you can use `nullptr`. That keyword stands for a null pointer variable with type `std::nullptr_t`. The `nullptr` is convertible to **any pointer** type and to `bool`.

```
const int *x = nullptr;
```

8. When using `union` for a mixture of structures that start the same way, there is a guarantee in C (and C++) of an identical physical mapping of components of a structure type from the beginning if they are the same.

9. In C and C++, there are the following guarantees for components of the variable with structure type (`struct`):

    - The components (members, fields) of the variable with structure type obtain addresses in *ascending order* as they are defined in the structure type.
    - The address of the first component is the same as the address of the beginning of the structure. It is regardless of whether the computer is big-endian or little-endian, where the program will run.

10. Structs are not allowed to perform comparisons with `==` or with `>`. One of the fundamental natures of this restriction in C (and in C++) is that, for objects, there may be holes in their memory layout that are filled randomly.

11. In C++, for the definition (not just declaration) of a variable in global scope, you can use `extern int a = 0;`. But in fact, `extern` can be ignored because according to (7.11.6, C++2003):

    > "A name declared in a namespace scope without a storage-class-specifier has **external linkage** unless it has internal linkage because of a previous declaration and provided it is not declared const. **Objects declared const and not explicitly declared extern have internal linkage**."

12. A compile-time string literal in C (and in C++) is statically allocated so that it is safe from a language point of view to return a pointer to a string literal from a function.

New Operator

In C++, before the introduction of the exception mechanism, the `new` operator returned 0 when the memory allocation failed.

In the C++ standard, `new` by default throws a `std::bad_alloc` exception. As a rule, striving for similarity to the standard is best. Better to modify the program to catch `bad_alloc` rather than check the return for 0. In both cases, doing anything other than throwing an error message is not easy on most systems. See paragraph 5.3.4. Subparagraph 13: http://www.ishiboo.com/~nirva/c++/C++STANDARD-ISOIEC14882-1998.pdf

- The new(std::nothrow) available in standard C++ does not throw an exception.
- Regular new - throws an exception.

Placement New

Essentially, there are several variations for "exotic" usage of the `new` operator:

1. Placement `new`. Creation of an object, but using the already prepared address space. If the implementation needs to store some meta-information, then it can be the case that `b != address`. Example:

```
#include <new>
int *b = new(address) int(init_value);
```

2. Overload operator new as a new global function in the global namespace. Example:

```
void* operator new(size_t sz) {return a.allocate(sz);}
void operator delete(void* ptr)
```

3. Overloading `new` with custom parameters. The first argument to the operator is the size in bytes and is calculated automatically via `sizeof` by the C++ compiler. Then, there is a list of arguments that you decide clients should pass. Example:

```
void* operator new(size_t sz, Arena& a, float b)
{ return a.allocate(sz);}

new(arg2, arg3) SOMETYPE()
```

4. Operator overloading in a class. You can define new/delete within a class.

> It's good practice to make new/delete `static`. However, the operator will be implicitly static, even if static is not explicitly specified.

Pseudo Destructor

When you need to call the destructor explicitly, and when you understand what you're doing, you can call in the C++ destructor via using **pseudo-destructor**.

Moreover, the **pseudo-destructor** can be virtual. Of course, you need to do that in very rare cases.

```
#include <stdio.h>
class A {
public:
   A()          {printf("A()" "\n");}
   virtual ~A() {printf("~A()" "\n");}
};

class B: public A {
public:
   B()           {printf("B()" "\n");}
   ~B()          {printf("~B()" "\n");}
};
int main() {
  A* a= new B;
  a->~A();
}
```

Aggregates

Formal definition from the C++ standard (C++03 8.5.1 §1): *An aggregate is an array or a class (clause 9) with no user-declared constructors (12.1), no private or protected non-static data members (clause 11), no base classes (clause 10), and no virtual functions (10.3).*

Aggregate types are unique in that objects of such types can be initialized in C++98/03 using the language's built-in curly brace syntax, just as structures are initialized.

POD or Plain Old Datatype (C++03)

An aggregate class is called a POD if it has no user-defined copy assignment operator and destructor and none of its non-static members is a non-POD class, an array of non-POD, or a reference.

So, finally, the POD type should contain:

- No user-declared constructors
- No private or protected non-static data members
- No virtual functions
- No user-defined copy assignment

And additionally:

- No base classes
- No destructor

If you want to write a more or less portable dynamic library that can be used from C and even .NET, you should try to make all your exported functions take and return only parameters of POD types.

The lifetime of objects of the non-POD class type begins when the constructor has finished and ends when the destructor has finished. For POD classes, the lifetime begins when storage for the object is occupied and finishes when that storage is released or reused.

For objects of POD types, it is guaranteed by the standard that when you `memcpy` the contents of your object into an array of `char` or `unsigned char` and then `memcpy` the contents back into your object, the object will hold its original value.

As you may know, it is illegal (the compiler should issue an error) to make a jump via `goto` from a point where some variable is not yet in scope to a point where it is already in scope. This restriction applies only if the variable is of non-POD type. It is guaranteed that there will be no padding at the beginning of a POD object.

Standard Layout (From C++11)

C++11 introduces a relaxed POD type definition - standard layout types. To have a standard layout, the following rules should be satisfied for your type:

- No virtual functions in your class/struct
- No virtual base class/struct
- No two base classes of the same type
- No restriction for static member functions and static members

However, in contrast with POD, it can contain:

- Zero or more base classes, but they should be standard-layout class types
- All data members should have the same access control
- All data members are defined in the most base class or most derived class.

If you want to operate with classes that allow work in low-level byte representation, the class characteristic (type traits) in which you should be interested is trivially_copyable.

Built-in Type Conversion

Before going into technical details about type conversion, let us be honest - it's hard to remember them. So, it is possibly better to observe the big picture first:

```
The general requirement when converting integer types is the mathematical
equivalence of the source and target values.
```

Now, let's go into the technical details.

Prohibited Conversions

1. Converting a pointer to a function to a pointer to data, and the other direction, is not allowed in C and C++.
2. Built-in conversion to a `struct`, or to the `union` is not allowed.
3. C++ treats `enum` as distinct from each other and from integer types as well.
4. In C, implicit conversion from integer to enumerated types is allowed because in C, `enum` is a synonym of `int`. In C++, it is prohibited.
5. In C and C++, implicit conversion from enumerated to integer types is allowed.

The Sequence of Type Conversions Rules in C and in C++

1. Trivial transformation. Conversion to identical types. A conversion from "function ..." to "function pointer ...."
2. If an overflow occurs during conversion to a `signed` type, then the value is considered overflowed and technically **undefined**.
3. If an overflow occurs during conversion to an `unsigned` type, the final value equals the "unique value" mod 2^n of the result. When using two's complement presentation, converting to/from signed to unsigned integers of the same size does not require any bit change.
4. If the final type is shorter than the original and both types are unsigned, the conversion can be performed by discarding the appropriate number of the most significant bits. The rule is also applicable to integer types in 2's complement notation.

5. When converting from float values to `int`, the final value should be equal to the initial value if possible. The non-zero fractional part is discarded. (The result is **undefined** if the value cannot even be approximated)

6. In C, conversion to floating-point types is possible only from arithmetic types. During conversion from `double` to `float`, the final value must equal one of the two values closest to the original value. (The choice of rounding is implementation-dependent)

7. If it is impossible to convert from double or int to float, then the value is **undefined**. (Example: If the range of the target double type does not match)

8. Conversion from the type array of type `T` to a pointer to type `T` is performed by substituting the pointer for the first element of the array.

9. A value of any type can be converted to `void`.

10. Conversion to `void *` and back guarantees the restoration of the original pointer value.

11. In C, `void *` can be **implicitly** converted to a pointer to any type. In C++, **an explicit cast** is required. (Appendix C, 4.10, C++ 2003 standard)

12. On the operands of unary operations, the ordinary unary conversions described next are performed. The goal is to reduce the number of arithmetic types. Here they are:

- An array of type T $\to$ pointer to the first element. However, it is not applied to arguments of the `operator &` and `sizeof` operators.
- Function $\to$ function pointer.
- Conversions from an integer type of rank below int $\to$ to `int`.
- Conversions from unsigned integer types lower than `int`, `int` represent all values $\to$ values are cast to integers.
- Conversions from unsigned integer types lower than `int`, but `int` does not represent all $\to$ values, are cast to `unsigned int`.

13. On the operands of a binary operation, the usual unary conversions are performed separately for each argument, and after that, the regular binary conversions are applied.

14. If some operands of the binary operator have the type `long`, `double`, or `float`, and the second operand has a lower rank, then it is cast to the type with the highest rank.

15. If both operands are unsigned, then both are cast to a higher-rank unsigned type.

16. If both operands are signed, then both are cast in the signed type of the higher rank.

17. Tricky part. Unsigned operand and signed type operand of higher rank $\to$ signed type.

18. Tricky part. Unsigned operand and lower-ranked signed operand $\to$ unsigned type.

19. If the prototype is controlled by an ellipsis `...`, i.e., the function obtains varying argument numbers, then:

- The usual unary conversions are performed on the operands.

- `float` is always promoted to `double` (the `float` is not converted into a `double` if there is no ellipsis and the call is fully prototype-driven)

Namespaces

Basics about Namespaces

The namespace is a mechanism for reflecting logical grouping. If some declarations can be combined according to some criteria, they can be placed in the same namespace to reflect this fact.

Namespace Advantages

- Logical structure reflection
- Avoidance of name conflicts
- Express a coherent set of tools
- Prevent users from accessing unnecessary tools
- Do not require significant additional effort when using namespace functionality

Namespace Disadvantages

- Waste of time analyzing the assignment of objects (types, functions) to different namespaces
- Various additional nuances:
    - A local variable or a variable declared via `using` hides external variables within the block of visibility.
    - When libraries that declare many names are made available through the `using namespace` directive, it is important to understand that unused name conflicts are not considered an error if they are not used.
    - Elements of the same namespace can be in different files.

Namespace Lookup Rules

A namespace is a named scope. Unlike a class definition, a namespace is open to new elements being added to it.

The `using` directive applies more to namespaces than to classes, even though there are some use cases when `using` is applied in class definitions.

1. If the function is not found in the context of its use, then an attempt is made to search in the namespace of the arguments. This rule exists to facilitate not polluting the namespace.

```
namespace NameSpace {
    struct Type {};
    void func(Type x)
    {}
}

int main() {
    func(NameSpace::Type());
}
```

This mechanism is called an Argument-Dependent Lookup (ADL). It's useful, for example, when calling some overridden operator on your type in a situation where you decide to define an operator in the namespace in which your type is defined itself.

Some nuances I've come across: https://stackoverflow.com/questions/45713667/unqualified-lookup-in-c. Various nuances of working with namespaces are covered in ([1] B.10, page 924). Below, we will take a look at the main one.

2. A locally declared name in the local scope and a name declared with a `using` directive **hide a non-local declaration**.

```
#include <iostream>

int i = 1;
int main() {
    using ::i;
    {
        int i = 2;
        std::cout << "local: " << i << '\n';
        std::cout << "global: " << ::i << '\n';
    }
    return 0;
}
```

3. Local name declaration takes precedence over the name imported from the namespace. The global declaration does not take precedence over variables imported from NS::*.

4. Collisions of unused names are not treated as errors.

5. Global names are in the "global namespace".

> Importantly, "global namespace" is just one more namespace. It differs from all namespaces (including the unnamed one). The global namespace differs from those defined through the namespace only because it is not necessary to write its name. It's only worth thinking about when you need to use `::global_var` when you have a problem (3). The operator `::` stands for scope extension. With this construction, you will **always look first in the global namespace** and then in the namespaces imported into the global namespace (including unnamed namespace), but not from the local scope.

```
#include <iostream>

int a = 1;
namespace
{
    int c = 23;
}
int main()
```

```
{
    int a = 23;
    int b = 2;
    std::cout << ::a;
    std::cout << ::c;
    //std::cout << ::b;  // compile-time error

    return 0;
}
```

6. If the name is declared in the enclosing scope or the current scope, then the name can be used without problems, without a full qualifier.

7. Continuous repetition of a qualifier distracts attention. Verbosity can be eliminated using the following constructions:

   7.1. **Using Declaration**. Creating a synonym for a variable in a namespace through `using NS::x;`.

   7.2. **Using Directive**. Create synonyms for all variables in a namespace by `using namespace NS;`.

8. Placing 7.2 inside another namespace opens up the way to combine/mix names from different namespaces.

9. Creating an unnamed namespace implies auto-generating its name by the compiler and insertion `using namespace GEN_NAME;` to the source file with that unnamed namespace.

10. Names explicitly declared in a namespace and which have been made available by using declarations, i.e., via `using NS::x;` take precedence over names made available via using directives, i.e., `using namespace NS;`

    ```
    #include <iostream>

    namespace A
    {
        int std;
    }
    using namespace std;

    int main()
    {
        using A::std;
        std = 1;
        return 0;
    }
    ```

11. Namespaces can be nested. To create an alias, you can use a construct like the following (See documentation):

    ```
    namespace AA = NameSpace::NameSpace2;
    ```

Example:

```
#include <iostream>

int i = 1;

using namespace std;
using std::cout;
namespace STD = std;

int main() {
    std::cout << "global: " << i << '\n';
    STD::cout << "global: " << ::i << '\n';
    return 0;
}
```

12. Namespace Search Rules in case of using nested namespace, by example. Next example with a variable `i` was taken from ANSI ISO IEC 14882, C++2003, 3.4.1, (6) (page 30):

```
namespace A {
  namespace N {
    void f();
  }
  int i=1;
}

void A::N::f() {
    i = 5; // access i unqualified
}

int main()
{
    return 0;
}
```

Because you access `i` by name, it implies execution of an unqualified name lookup, because a name that does not appear to the right of a scope resolution `operator ::`

The following scopes are searched for a declaration of `i`:

1. Innermost block scope of `A::N::f`, before the use of `i`
2. Scope of namespace `N`
3. Scope of namespace `A`
4. Global scope, before the definition of `A::N::f`

13. One subtle addition to function names. Function names obtained from ADL (Argument-Dependent Lookup) are looked up in the namespaces of their arguments in addition to the scopes and namespaces

considered by the usual unqualified name lookup.

Examples of Using Keyword using

```
using std::cout;                            // Make cout available without
qualification
using namespace std;                        // Make all names in std available
without qualification
using Big = unsigned long long;             // Defines Big as a type alias

typedef unsigned long long BigWithTypedef; // Defines BigWithTypedef as an alias
for long long. Typedefs in C++20 are obsolete.
```

Exceptions

Basics about Exceptions

When a program is constructed from separate modules, and especially when these modules are in independently different libraries, it is convenient to divide error handling into two parts:

- Generation of information about the occurrence of an error situation that cannot be resolved locally
- Handling errors found elsewhere

> Error-handling code can be shorter and more elegant using a return value, but this solution does not scale well.

Generally speaking, separating error-handling code from "normal" code is a good strategy. Throwing an exception may leave the object in an invalid state. However, throwing an exception can be a source of memory and other resource leaks. It's best to rely on the properties of constructors and destructors and their interactions with exception handling to deal appropriately with objects.

Escape from a block by throwing an exception cleans up all created local, automatically allocated objects in reverse order of creation. Writing correct exception-safe code using explicit tries can be a difficult task.

Extra about Exceptions

General aspects:

- You can group exceptions by inheritance relation.
- Exiting a destructor by throwing an exception is against the requirements of the standard library.
- The process of calling destructors for automatic objects constructed on the path from a try block to a throw expression is called **"stack unwinding"**.
- If an exception is thrown but not caught, `std::terminate` is called. You can set your behavior with `std::set_terminate`.
- It is possible to rethrow an exception with `throw;`. This rethrows the existing exception object.
- When an exception is thrown in a constructor, the object's destructor is not called.

- The basic error classes are `exception`, `logic_error`, `runtime_error`. Some other classes: `bad_alloc`, `bad_cast`, `bad_typeid`, `bad_exception`, `out_of_range`, `invalid_argument`, `overflow_error`, `ios_base::failure`.

Technical details:

- Exceptions at the time of generation are copied. The `const` modifier in the catch block does not affect anything. However, the presence of a `T&` or `T` type signature is affected. The latter causes the copy constructor to execute. For `throw T();` you can't see the copy constructor run in VS 2012. But it can be seen for:

```
{T e; throw e;}
```

> **How is memory for exception objects allocated?** Unfortunately, the answer to it is pretty vague, and it depends on the toolchain (see that cppreference note): "On the implementations that follow Itanium C++ ABI (GCC, Clang, etc.), exceptions are allocated on the heap when thrown (except for bad_alloc in some cases), and this function creates the smart pointer referencing the previously allocated object. On MSVC, exceptions are allocated on the stack when thrown, and this function performs the heap allocation and copies the exception object when needed."

- If a destructor is called during stack unwinding, exits via throwing an exception on its own, terminate is called (C++2003, 15.5.1). So, destructors should generally catch exceptions and not let them propagate out of the destructor.
- If an exception is thrown in the destructor during the call to the destructor during exception handling, then this is considered an error in the exception handling mechanism, and `std::terminate()` is called. To distinguish the behavior of executing a destructor due to the normal call of the destructor or during stack unwinding, you can use `uncaught_exception()` in the destructor.
- If a function tries to throw an exception it didn't declare, it will result in a call to `std::unexpected`, which by default calls `std::terminate` (p.429, B. Stroustrup, special edition)

```
int f();                /* Can throw any exception */
int f() throw();        /* Throw no exceptions */
int f() throw(x2, x3); /* Throws only exceptions x2, x3 */
```

Starting from C++11, it's possible to use an empty exception specification `throw()` defined in an alternative way via `noexcept`.

```
void g1(int) throw() {}
void g2(int) noexcept {}
```

If you see a `noexcept` in a function's header, you can be sure that this function will never throw an exception.

Construction of the interception of exceptions from the initialization list. Example:

```
#include <stdio.h>

class C
{
public:
  C() try : a()
  { puts("C()");  }
        catch(...)
  { puts("WOW!"); }
  int a;
};

int main() {
  C c;
  return 0;
}
```

Overloading

Functions and Operator Overloading Precedence

The function's return type is not part of the function's signature. To decide which function overload to use, the compiler looks only at the number and types of the function parameters and arguments.

1. Firstly, the compiler tries to find a function with exact type matching, or the matching is achieved by trivial conversion (array name to a pointer, function name to function pointer, type `T` to `const T`).
2. At a second level, the compiler tries to make correspondence by the promotion of integral types and the promotion of real numbers. (`char` to `int`, `float` to `double`).
3. Correspondence achieved by (a) standard conversions (`int` to `double`, `double` to `int`); (b) pointers to derivative classes are cast to base classes; (c) Pointers to arbitrary types to pointers to `void*`.
4. Correspondence achieved with user-defined transformations.
5. Correspondence due to ellipsis `...` in the function definition.
6. **Important for C++11 and later:** Because C++11 introduced a new fundamental type of reference as `A&&`, the rules for function overloading have been upgraded a bit. Overload the function with references in form `A&&` has **priority** under constant reference `const A&`.

If a match can be achieved in **two ways** at the same criteria level, the overloading is considered ambiguous and is considered to be a compile-time error.

In the context of overloading class member functions, there is one problem when we want to take into account not only members for the current class but also member functions of a base class. The C++ treats differently the class scope of the base and derived class in the context of identical function names (and in particular in context of *overloading*).

A very popular "fix" of the standard (and correct) behavior of the C++ compiler, not trying to find an overloaded function from base class is the following:

```
class B  {
public:
  void f(int i) { std::cout << "f(int)\n"; }
};

class D: public B {
public:
    // Fix "strange" behavior
    using B::f;
    void f(double d) { std::cout << "f(double)\n"; }
};
```

Starting from C++11, the `using` keyword is also used for Base Constructor inheritance. Please check section "Miscellaneous Features of C++11" of this document for a description.

Template Function Overloading

Template function overloading searches for a set of suitable specializations according to steps (1-4) described below:

1. All specializations that can potentially be called.
2. If any specialization is more specialized, the less specialized is discarded.
3. Overloading allowed for functions from steps 1-2 and for regular functions.
4. If a regular function and a template function match equally well, the regular function takes precedence.

The call is considered an error if the function passed 1-4 cannot be found.

Resolving the Overloaded Binary Operator for x(op)y

1. If `x` has type `X`, then try to find out if the operator is defined as a member of `class X` or a `base class of X`.
2. Look for the operator declaration in the context of `x(op)y` expression.
3. If `X` is declared in `namespace N`, take a look for the operator declaration in `namespace N`.
4. If the type of `y` is `Y` and `Y` is declared in `namespace M`, look for the operator declaration in `namespace M`.

Operators Overloading Rules in C++

Please check the details in the Standard. C++ 2003. p. 233. 13.5.1 - 13.5.7.

According to the Language standard, operators **cannot** be overloaded as static methods of a class. You can remember it, because it's a bit illogical type `a A::+ b`.

It is possible to overload operators in C++ only:

- As a function
- As non-static class methods
- Some operators `=`, `[]`, `->` can be overloaded as a non-static class member function only.

Next, you cannot overload the following operators:

- `.`
- `::`
- `.*`
- `->*`
- `sizeof`
- `typeid`
- `?:`

Keyword typename

The typename keyword must be used in three tasks.

1. Replacing the keyword class with the word `typename` in the argument type declaration for a template class/function/method.

```
template <typename T> struct S{};
```

2. Accessing type names through the scope of the template argument type/class.

```
template <class T> struct S {
  typename T::SomeType a;
}
```

   Comment by B. Stroustrup *"In some cases, a smart compiler could guess, but in general, it's not possible."*.

3. The `typename` keyword is required if the type name depends on the selected template parameter type of the function or class indirectly.

```
template <class T> T findMax(const std::vector<T>& vec)
{
  typename std::vector<T>::const_iterator max_i = vec.begin();
  for (typename std::vector<T>::const_iterator i =
  vec.begin() + 1;
  i != vec.end();
  ++i)
  {
    if (*i > *max_i)
      max_i = i;
```

```
    }
    return *max_i;
  }
```

There is also a very detailed description of the `typename` keyword available here: http://stackoverflow.com/questions/610245/where-and-why-do-i-have-to-put-the-template-and-typename-keywords.

Class Constructors and Destructors

Logic behind executing Constructors

The order of initialization of a class object in C++ is described in C++ Standard - ANSI ISO IEC 14882 2003; 12.6.2, p.230. The order of initialization of classes and execution of constructors is as follows:

1. Depth-first, left-to-right constructors of **virtually inherited** classes whenever they are located in the inheritance tree.
2. Execution of constructors of directly base classes in the order from left to right from the class description (and **not in the order specified in the initialization list**).
3. Initialization of class members in the order specified in the class description (and **not in the order specified in the initialization list**).
4. Execution of the function body of the constructor.
5. The constructor also introduces an implicit conversion. To suppress implicit conversion, the constructor must be declared with the `explicit` parameter ([1], p.333).
6. In C++, when using (i) custom conversions via constructors without an `explicit` keyword; (ii) defined `type conversions` - then only one level of implicit conversions is allowed.

Example:

```
#include <stdio.h>

class A {
public:
  A(){printf("A()\n");}
};

class B: virtual public A {
public:
  B(){printf("B()\n");}
};

class C: /*virtual*/ virtual public A {
public:
  C(){printf("C()\n");}
};
```

```
class Branch {
public:
  Branch(){printf("Branch()\n");}
};

class D: public Branch, virtual public B, virtual public C {
public:
  D(){printf("D()\n");}
};

int main() {
    D d;
    return 0;
}
```

Surprisingly, it will not be Branch() that will be printed first, but the output will be like this:

```
A()
B()
C()
Branch()
D()
```

Logic Behind Executing Destructors

When processing the destructor, identical actions behind the logic of the constructor's execution are performed but in reverse order.

1. Execution of the destructor body of the class destructor.
2. Work out the body of the destructor for class members.
3. Destructor of direct non-virtual base classes.
4. If the object is the object of the "deepest" class in the inheritance graph, then virtual base destructors are called.

Deleting Object of Incomplete Type

Deleting an object with an incomplete type.

```
class My;
My* ptr;
delete ptr; // undefined behavior for incomplete types
```

For POD and an object without a destructor, something like C runtime free will be done, which does not need to know the object's size. In this case, you're lucky, and you can typically delete dynamically allocated objects, but in general, it results in undefined behavior. (5.3.5 Delete C++2003).

Generate and Suppress the Generation of Special Class Members

You can explicitly force the compiler to generate default code for a method for which this behavior can take place through `=default` construction. It can be used for:

a. Change the normal accessibility level or virtualize them. For example:

```
// Make destructor virtual
class MyClass {
public:
  virtual ~MyClass() = default;
};
```

b. For un-suppressing implicitly generated special member functions.

```
class MyClass
{
public:
MyClass(const MyClass&);
// copy constructor prevents implicitly-declared
// default ctor and move constructor

MyClass() = default;
MyClass(MyClass&&) noexcept = default;
};
```

You can disable any function or class method by specifying `= delete`. Example:

```
// g callable with any pointer type
void g(void*);

// g uncallable with const char*
void g(const char*) = delete;
```

Some Class Special Members (since C++11)

```
#include <iostream>

class X {
public:
  // "ordinary constructor": create an object
  X(int a)
  { std::cout << "X::X(int)" << std::endl; }

  // Default constructor
  X()
```

```
    { std::cout << "X::X()" << std::endl; }

    // Copy constructor
    X(const X&)
    { std::cout << "X::X(const X&)" << std::endl; }

    // Move constructor
    X(X&& arg) noexcept
    { std::cout << "X::X(X&&)" << std::endl; }

    // Copy assignment: Clean up the target and copy
    X& operator=(const X&)
    { std::cout << "X::operator=(X&)" << std::endl; }

    // Move assignment: clean up the target and move out from arg
    X& operator=(X&& arg) noexcept
    { std::cout << "X::operator=(X&&)" << std::endl; }

    // Destructor: clean up
    ~X()
    { std::cout << "~X()" << std::endl; }
};
```

Now, let's discuss the rules of the C++ compiler for generating special members.

First, every class should have a constructor to construct objects of the class and as a consequence:

> Rule - 1: If the class has no user-declared constructors, the compiler will try to generate a default constructor.

The existence of a user-defined destructor means that the class has special logic to destroy the object, and as a consequence, the class may have some invariants in the class. In this case, *it's dangerous for the compiler to generate a move constructor*:

> Rule - 2: If a class has a user-declared destructor, the compiler will not generate the move operator and move assign operation.

On another side, if the user declares copy constructor or `operator=` by himself, it also means that some special logic is required for the move:

> Rule - 3: If a class has a user-declared copy constructor or `operator=` compiler will not generate the move operator and move assign operator.

The C++11 will try to generate a move if no user-defined copy or move and no user-declared destructor(dtor). The same logic is used for generating copy operations:

> Rule 4: If there is no user-declared move constructor and move assign operator, the compiler will generate a copy constructor and copy assign operation.

To experiment with these rules, you can play with the following code snippet:

```
```

```cpp
#include <iostream>

class X {
public:
    // "ordinary constructor": create an object
    X(int a) {
        std::cout << "X::X(int)" << std::endl;
    }

    // Default constructor
    X() {
        std::cout << "X::X()" << std::endl;
    }

    // Copy constructor
    X(const X&) {
        std::cout << "X::X(const X&)" << std::endl;
    }

    // Move constructor
    X(X&&) noexcept {
        std::cout << "X::X(X&&)" << std::endl;
    }

    // Copy assignment: clean up the target and copy
    X& operator=(const X&) {
        std::cout << "X::operator=(X&)" << std::endl;
    }

    // Move assignment: clean up the target and move
    X& operator=(X&&) noexcept {
        std::cout << "X::operator=(X&&)" << std::endl;
    }

    // Destructor: clean up
    ~X() {
        std::cout << "~X()" << std::endl;
    }
};

class Y: public X {
public:
    ~Y(){}
};

int main(int argc, char** argv) {
    Y a;
    Y b(std::move(a));
}
```

There is one blog post resource that shows when the compiler generates special member functions in C++11. For convenience, we present a table from this blog post here:

Initialization

C++ Variable Initialization

1. There are a lot of types of initialization in C++.

```
const int v1(expr);   // direct initialization
int v2 = expr;        // copy/assignment initialization
int arr[] = {1,2,3};  // brace/bracket/uniform initialization (form-1)
int a {15};           // brace/bracket/uniform initialization (form-2)
int count(4);         // functional notation for initialization (in standard
denoted as direct initialization)
int z{};              // zero initialization

class X {
public:
  // Brace initialization from C++11 can be used to initialize arrays in the
initialization list
  // In C++03/98, such initialization was impossible
  X()
    // brace/bracket/uniform initialization (form-2). [OK].
    : x {1, 2, 1}
    // brace/bracket/uniform initialization (form-1). [Compile Error].
    // : x = {1, 2, 1}
  {}
private:
  const int x[3];
};
```

For detailed information about different rules of initialization, please use cpp reference documentation.

2. Static variables are initialized to zero for the corresponding type.
3. If the initialization list is empty, the array and structure variables are initialized to zero.
4. A reference in C++ shall be initialized to refer to a valid object or function. In particular, a null reference cannot exist in a well-defined program.
5. Types of the most popular initialization types in C++:

- default_initialization
- value_initialization
- zero_initialization

5. If, for an array or structure, the number of initialization values is less than the dimension of the array or structure, the rest of the elements are initialized with the default value for the corresponding type (as in the case of initialization of static variables).
6. In C++98/03, a `union` cannot have as a member an object with a user-defined constructor.

std::initializer_list

In C++98, constant member arrays are impossible to initialize through the initialization list in the class constructor, but in C++11, there is an idea to bring bracket initialization to everything.

```
int a[] {1,8,8}
```

Here, `{1,8,8}` is an initialization list. This construction implicitly casts the right-hand side to `std::initializer_list<T>`. In constructors with initialization lists, the list argument has priority during overloading.

Type deduction in template functions does not work for `std::initializer_list` type. But for auto, it works fine. **It is one place where auto != template**.

Example:

```
// gcc -x c++ -std=c++11 t.cpp
#include <stdio.h>
#include <initializer_list>

template <class T>
void f1(T a) {}

void f2(std::initializer_list<int> a) {}

int main() {
  f2({1,2,3});
```

```
    //f1({1,2,3}); // DOES NOT COMPILE
    auto f1arg = {1,2,3};
    f1(f1arg);

    return 0;
}
```

The `std::initializer_list` stores initializer values in the underlying array. This type has the following member functions:

- **size**: The number of elements in the array.
- **begin**: Pointer to the first array element.
- **end**: Pointer to a one-beyond-last array element.

7. In ([1], 10.4.6.2) for C++2003: "*You can initialize a class member that is a static constant of an integral type. If and only if you do so, you need to declare this member once somewhere.*". Somewhere in this context, it means some compilation unit. Practically, in MSVC the fact that variable is integer (not float), and constant is important.

Various Constant Flavors

const (C++03)

The keyword `const` prevents `const` objects from getting mutated. But `const` can be initialized by something that is not a constant expression.

The `const` member function cannot change the state of the object.

constexpr (C++11)

Documentation: cpp reference details.

The `constexpr` is a modifier of a function or a variable.

The `constexpr` indicates that it *should be possible* to evaluate the function or expression at compile time if given constant expressions as arguments.

However, it can be run at compile time and runtime.

The main idea of B. Stroustrup is that it brings type-rich programming at compile time. The deep reason that includes this in the language is that many communities ask B. Stroustrup to have something that will make table look-up easier in languages. And because it's nearly impossible to be faster than a table lookup, this concept can make sense.

C++11 `constexpr` functions had to put everything in a single return statement. In C++14 `constexpr` functions, one does not necessarily have to put everything in a single return statement. Example:

```
constexpr int sum(int a, int b)
{
```

```
    return a + b;
}
```

```
constexpr int a = 12;
constexpr int sum(int a, int b) { return a + b; }
```

Also, importantly, `constexpr` is implicitly thread-safe.

The `constexpr` function can be run in compile time and runtime. In C++2020, there is a way to distinguish between compile-time evaluated code and code that is executed in runtime by using `std::is_constant_evaluated()` (available since C++20 is_constant_evaluated) inside the function with the `constexpr` modifier.

The `constexpr` functions requirements:

- It has *the potential* to be run at compile time
- It resolves dependencies at compile time
- It does not have side effects (pure in the mathematical sense)
- It does not have `static` or `thread_local` variables inside

consteval (C++20)

The `consteval` is a modifier of a function only.

The `consteval` expressions generated an immediate function, which **can only be run at compile time.**

If `consteval` function cannot be run during compile time, it leads to a compile-time error. Every call to `consteval` generates a constant expression that is executed at compile time. The `consteval` function has the same requirements as `constexpr` functions:

- It has *the potential* to be run at compile time
- It resolves dependencies at compile time
- It does not have side effects (pure in the mathematical sense)
- It does not have `static` or `thread_local` variables inside

constinit (C++20)

The order of initialization of usual `static` variables from different translation units in C++03/11 is undefined. And even more, in C++03/11, `static` variables initialization can happen in *compile-time* or in *run-time*.

The `constinit` is a modifier of a variable with static storage duration.

- The `constinit` guarantees that a variable with static storage duration (global variables and static variables inside the functions) is initialized at *compile time*.
- The variable can be mutable and is not necessarily a `const`, but initialization must be performed at compile time.
- The `constinit` modifier can only be applied to a variable with *static* storage duration.

```
```

```
int sqrRuntime(int x){return x*x;}

constexpr int sqrCompileOrRunTime(int x){return x*x;}

consteval int sqrCompileTime(int x){return x*x;}

static constinit double kPi = 3.14159256;
```

Compute Optimization Relative Information

Return Value Optimization

In a return statement, a compiler is allowed to apply the return value optimization (RVO), provided the returned name is the name of a locally defined automatic variable.

Inline Function Call

Working with inline functions is described here: https://isocpp.org/wiki/faq/inline-functions. Example of how to work with inline functions:

```
// inline.h
#include <stdio.h>

void f();

inline void f() {
  printf("F inline");
}
```

And

```
// main.cpp
#include "inline.h"

int main() {
  f();
  return 0;
}
```

Reasons why the compiler can not inline a function:

- During the compilation of a source file, the definition of a function is not available (unless you do not use whole program optimization in modern compilers).

- The compiler performs a cost-benefit analysis to decide whether to inline a function or not. Specifically, the inlining function has some **cost**, and compilers have a threshold for **inlining**. The underlying reason is that if the inline code affects the size of the code, then it affects performance as well.
- The compiler designers may have problems inlining recursive function calls (except for tail recursion).

Force Inline Function Call

One way to force the inline of some function change `inline-threshold` in your toolchain is by using the `forceinline` compiler extension. Example:

```
// MSVC
__forceinline static int max(int x, in y) {
  return x > y ? x : y; // always inline if possible
}

// GCC/CLANG
inline __attribute__((always_inline)) int max(int x, int y) {
  return x > y ? x : y; // always inline if possible
}
```

The `__forceinline` due to the first two underscores is a compiler extension. This asks the compiler to insert a copy of the function body into each place the function is called. This can make the program faster by eliminating the overhead of function calls, but it can also increase the code size. The `__forceinline` overrides this analysis.

You should use `__forceinline` only when you are sure that inlining the function will improve the performance of your program. This compiler instruction is a compiler extension and available for example via:

- In GCC instead of `inline` use `inline __attribute__((__always_inline__))`.
- In MSVC, instead of `inline` use `__forceinline`.

However, `__forceinline` still does not guarantee to inline; e.g., in MSVC, there are various reasons why inline can not be done (See __forceinline in msvc)such as:

- Recursive functions with a relatively big depth of recursion
- For a virtual function called really virtually
- Function is invoked via a function pointer

The underlying reason for using **force inline** is that compilers use only heuristics to decide to perform inlining, and sometimes these heuristics are wrong.

Allowable Reformulations

In C++ and C, the order in which subexpressions are evaluated in constructions like `A(F(), G())` is not defined. The compiler is allowed to evaluate the operands of a binary operation in an arbitrary order and perform optimizations depending on the associativity and commutativity of the operation.

Compiler Implementation Relative Questions

What does `A(*B)` mean? It can be declaring a variable or calling a function. This ambiguity makes the C grammar formally context-sensitive and not LALR(1). In C++, access control is checked later than ambiguity.

```
class A {
public:
  int a;
private:
  int a;
};
A z;
z.a = 1; // Compilation error
```

std::aligned_storage

In C++11, there is a type aligned_storage. It can be used for aligned but uninitialized memory buffers. Example:

```
#include <new>
#include <type_traits>

struct X
{
    int a;
    int b;
};

int main()
{
    typedef std::aligned_storage<sizeof(X), alignof(X)>::type XArrayType;
    XArrayType buffer;
    X* x = new (&buffer) X(); // construct X in buffer
    return 0;
}
```

- The `buffer` can hold an X object. The size and alignment are fine.
- Facilitates use of stack-based buffers (buffer is not on the heap)
- Like alloca in POSIX, except standardized
- Main limitation: the buffer size must be known during compilation.

Documentation: cpp reference aligned_storage.

Memory Aliasing and Restrict

Example:

```
void f(int* a, int* b)
{}
```

In C and in C++, when you deal with pointers, for example, in a function mentioned above, there are two possibilities:

(1) Pointers `a` and `b` refer to the exact location in memory, at the start of function execution or some moment during the runtime of function execution. It's completely legal for such a function signature.

(2) But in fact maybe pointers `a` and `b` never refer to the same memory location at all during function execution.

Without any extra help, the compiler has serious problems deciding what the case is for function `f`. And by default, the compiler assumes (1) case. Even though from a compiler optimization perspective, case (2) is far more favorable to deal with.

In C99, there is a way to make extra help for the compiler and say that, in fact, we are in situation (2), not (1); and specify pointers as `restrict` via the following syntax:

```
void f(int* restrict a, int* restrict b)
{}
```

C++ does not have such a keyword even up to C++20, but toolchains typically provide C++ extension via the `__restrict` extension (it is supported in GCC, MSVC, CLANG with the same name).

> The essence of `restrict` is described in [2,p.94], and here we repeat it shortly. Restrict means that within the scope in which such a pointer is defined, the named pointer is the only one way to access the object where the pointer is pointed. For function parameters in the form of pointers with `restrict`, it means that within the function scope, pointers `a` and `b` always point to different memory locations. An essential aspect of the `restrict` pointer is that due to [2,p.95] and the formal definition of `restrict` from C99, it would have an effect if the use of pointed objects as LValues, i.e., the memory in pointers is used for the *write*. In the case of using objects by `RValue`, the restriction does not impose any semantic restriction of memory that does not overlap where pointed objects are located. Because references are implemented as pointers at the level of ASM code generation, the same holds true for references.

But why is aliasing such a big deal? There are several general problems with it.

One way to explore problems is to use the tool OptView (https://github.com/OfekShilon/optview2) from Ofek Shilon. OptView is in the process of merging with the LLVM master. This tool is a step forward in the dialogue between the compiler and the person who creates the software. This section is based on the presentation LLVM Optimization Remarks - Ofek Shilon - CppCon 2022 in which the author presents OptView.

Clobbered by Store or Load

To visualize this effect, you can use the following code snippet and compile it with godbolt for clang with flags `--std c++20 -O3`:

```
#define CODE_SEC 2

#if CODE_SEC == 0
// First way to get rid of aliasing
void f1(int* out, const int & __restrict input_to_add, int n)
{
    for (int i = 0; i < n; ++i)
    {
        out[i] += input_to_add;
    }
}
#endif

#if CODE_SEC == 1
//The second way to get rid of aliasing
void f2(int* out, const int & input_to_add_, int n)
{
    int input_to_add = input_to_add_;
    for (int i = 0; i < n; ++i)
    {
        out[i] += input_to_add;
    }
}
#endif

#if CODE_SEC == 2
// Alliasing
void f3(int* out, const int & input_to_add, int n)
{
    for (int i = 0; i < n; ++i)
    {
        out[i] += input_to_add;
    }
}
#endif
```

To observe CLang compiler optimization problems with godbolt, please:

- select x86-64 clang compiler
- build with `--std c++20 -O3`
- select **Add New->Opt Remarks** in the web interface.

One good heuristic way to assess that reformulation of the situation with aliasing helps is a reduction in the assembly code size by a factor of x1.2 or x2.

Clobbered by Store/Load is a compiler problem that occurs due to aliasing in C++. The compiler protects itself from cases when there is a sequence of *writes*, and one of these writes can potentially overwrite read arguments for a function. A problem can occur with a variable from which the read happens through a reference or pointer. In example below it will happen if `{a[0],...,a[n-1]}` and `&input_to_add_` overlaps in memory.

There are several ways to resolve this problem:

1. One way to declare a pointer and reference is by `restrict`.
2. The second way is to copy the variable that causes aliasing into a temporary variable. It's bad for code readability but good for the compiler.
3. In C++, objects of different types will never refer to the same memory locations (objects are not aliased). In practice, it may not work. This solution is called *strict aliasing*.

Clobbered by Call

If a function obtains an argument by reference or pointer, then the potential object that you pass to a function can potentially be escaped. The technical term of this situation is *pointer escape*. Specifically, once a function obtains an argument by reference or pointer, the function can (potentially) store its address in some global state, and this value may be used in future calls.

Once the pointer is escaped, a lot of bad things can happen, unfortunately, e.g. some global function can read/write this state during calls, even if this global function has no arguments! If you have such a function and you want to optimize it, there are several things to do:

1. Make a promise that the function does not modify the global state (pure).

- In GCC and CLANG, the function attributes are introduced by the `__attribute__` keyword in the declaration of a function, followed by an attribute specification enclosed in double parentheses. See Function-Attributes in GCC. How to make it in GCC/CLANG:

```
int f(int& a) __attribute__((pure));
int f(int& a)
{return a + a;}
```

2. Make a promise that the function does not modify and even does not read the global state (const). In GCC and CLANG:

```
int f(int& a) __attribute__((const));
int f(int& a)
{return a + a;}
```

3. Make a promise that the address of the function argument does not escape from the function call(noescape).

In GCC, this function attribute is not supported. In CLANG, it is supported:

```
int f(int& a) __attribute__((noescape));
int f(int& a)
{return a + a;}
```

4. Other ways

Create a temporary object and transfer the temporary object into a function by reference. Temporary objects can also be useful in situations for moving load from loop invariant code. However, this is considered bad practice from the code-writing side.

5. About MSVC

One header-only library that contains various compiler-specific extensions is Hedley. Please take a look at this library in your projects.

Unfortunately, the MSVC does not provide *pure*, *noescape* function semantics.

It provides with __declspec(noalias) semantics similar to `__attribute__(const)`. Example:

```
__declspec(noalias) void multiply(float* a, float* b);
void multiply(float* a, float* b)  {
    a[0] *= b[0];
}
```

According to the Microsoft documentation, we can use `__declspec(noalias)` in the function declaration to indicate that the function does not modify memory outside the first level of indirection from the function's parameters. However, there are several differences between GCC/CLANG:

1. Grammatical difference. Grammatically in MSVC decl specifier `__declspec()` should come before the function declaration. In GCC/CLANG, the function attributes should come at the end of a function declaration.

2. Semantic Difference. There is a difference with the first level of indirection. The following example is not a valid GCC/CLANG code:

   ```
   void multiply(float* a, float* b) __attribute__((const));
   void multiply(float* a, float* b) {
       a[0] *= b[0];
   }
   ```

   The attribute((const)) syntax is used to indicate that the function results are entirely dependent on the provided arguments and do not mutate any external state. But `multiply` in this example modifies the value of *a, which violates the `const` attribute semantics in GCC/CLANG.

Popular Compiler Flags for Optimization

Below are compiler flags for GCC/CLANG, which can be useful for your code optimization:

- -O0 : Do not optimize
- -Og: Flag for debugging purposes
- -O1: Optimize
- -O2: Optimize even more.

- -O3: Aggressive optimization.
- -OS : Optimize to limit code size
- **-march=core2:** Tune for a specific architecture.
- **-march=native -mtune=native:** Automatic detection of current processors' features and tune for it.
- -ftree-vectorize: Automatic use of SSE.
- -funroll-loops: Loop unrolling.
- -ffast-math: Unsafe floating-point optimizations.
- **-fno-rtti**: Turn of RTTI (no dynamic_cast, typeid)
- **-fno-exceptions:** Turn of exception support from C++
- -flto: Turn on global program optimization (link-time optimization). LTO can help with aliasing and inlining theoretically; however, in practice, it is not always the case.

Several Principles for Code Optimization

- Sometimes recomputing is faster than saving/loading
- Avoid branches inside the internal loop
- Use local variables to expose independent computations
- You should decide what to use - function calculation or a table of precomputed values
- You can make several (independent) passes over a data structure or one combined pass
- Dense operations or operations that exploit sparsity

These principles have been taken from a course [12] at Cornell University and from other technical documents/guidelines provided by microprocessor vendors.

Another two important aspects of code optimization in C++:

- The `const` of references or pointers guarantees *nothing* from the code emitting point of view.
- The `private, protected` class fields garantees *nothing* from code emitting poit of view.

Using Static Const Variables to Avoid Copying

Declaring a variable as `static const` can improve performance by eliminating unnecessary copying.

When a variable represents a constant value that does not change and it is not dependent on the potentially recursive invocation of a function, using `static const` allows the compiler to store the value in fixed memory once.

While it might seem that declaring a local variable with the `const` qualifier achieves the same effect, there are key differences.

In the absence of whole-program optimization (even with optimizations like -O3), modern compilers lack global knowledge of potential recursive invocations of a function. Consequently, constant variables or arrays with automatic storage duration (const but without static) will still be allocated on the stack each time the function is called.

p.s. For more details listen the talk Fast and Small C++ When Efficiency Matters, A.Fertig, CppCon'24.

Compressed Pairs

Using a Compressed Pair when one of the objects has no data members. For a concrete example, see no_unique_address. This example has been mentioned in the talk Fast and Small C++ When Efficiency Matters, A.Fertig, CppCon'24.

Lambda Functions

Lambda expressions offer a convenient, compact syntax to quickly define callback functions. Basic syntax:

```
[] (int x, int y) { return x < y; }
```

Terminology

- `[]` - *lambda introducer* is the opening square brackets. It can not be omitted.
- `(int x, int y)` - *lambda parameter list* are parameters between round parentheses. Formally, in fact you can omit the empty parameter list for lambda functions without parameters and write: `[] {...}`. However, it may not be so good for readability.

> Closure in Math. In math, closure is (i) a set with its boundary (*Functional Analysis*) or (ii) a family function, which can be achieved via constructing all possible formulas under the basis function (*Discrete Math and Formal Languages*). The (ii) and (i) look the same - you explore the space completely via allowable steps.

- *Closure in C++* - in C++ *lambda closure* is a functional object (instance of the class) with the defined:

  ```
  ret_type operator()(<list of arguments>) const
  ```

C++ does not specify the closure type name, but you can automatically deduce it for a specific variable using `auto`. Example:

```
auto isOdd = [](int x) { return x % 2 == 1;};
```

The compiler creates the *closure object* during the parsing of the lambda expression. So essentially, Lambdas generate C++ classes with overloaded `operator()`.

You can refer to variables with static storage duration in Lambdas. They are available inside the closure. The technical term for this is that they are *captured*.

You can use lambda to capture values **by value**. In such a case, captured variables will be copied to a function object (closure). Example:

```
[captureVar1,captureVar2](int arg1){}
```

You can capture variables **by reference**. To do this, use `&`. In the context of lambda, `&` means capture by reference, not the dereferencing operator. Example:

```
[&captureVar1,&captureVar2](int arg1){}
```

Next, there exists a notation to capture all non-static variables by value or by reference:

```
// capture all not-static vars (which are used inside lambda) by value
[=](int arg1){}

// capture all not-static vars (which are used inside lambda) by reference
[&](int arg1){}
```

Next, a notation exists to capture all non-static variables by value or reference and specify something for other variables. Example:

- Capturing **all** non-static vars by reference
- Capture **Param2** by value

```
[&,Param2](int arg1){}
```

The important thing is that the capture default (`=` or `&`), if used, should always come first in the Lambda introducer. [4, p.752]. To capture specific variables by the value, you should use *only the name of the variable without the `=` prefix*. Maybe it is a bit confusing, but it's a language rule.

> The clause `[=myVar]` is invalid.

The lambda return type can be deduced if the lambda is one expression in C++11. Or you can explicitly specify it:

```
[=](int arg1)->trailing_return_type
{
  return trailing_return_type();
}
```

If lambda has more than one expression, then the return type:

- Must be specified via the trailing return type in C++11
- The compiler will deduce it from C++14 or later

A similar trailing syntax can be applied with `auto` and the usual `member` functions from C++11. You can capture only local variables, not member variables of the object. You can allow Lambda to modify captured values with the mutable modifier.

It's possible to add the keyword `mutable` to the definition of a lambda expression right after the *lambda parameter list*. Doing so causes the compiler to omit the `const` keyword from the function call operator of the generated class.

```
int x = 12;
auto a = [=]() mutable -> void  { x = 1; };
```

> Classical use case of `mutable` is to allow modification of the class data member even if the containing object is declared `const`.

Next, you can capture this pointer explicitly

```
auto a = [this]() ->void  { aa = 1; };
```

or implicitly, because default capture also captures `this` available.

```
auto a = [=]() ->void  { aa = 1; };
```

With the capture `this`, there are the following subtleties:

- Even though the lambda closure will be an object of a class, its function call operator **will have access to all protected and private members of the class** provided with `[this]` pointer to the object.
- The language does not provide the means to distinguish `this` of the function object and captured `this`. The **need "this" pointer** variable will be found automatically.

Example:

```
class X {
private:
    int x;
public:
    int f() {
        auto z = [this]() {return x; };
        return z();
    }
};
```

Even lambda is not a function pointer, and it is not an anonymous function, but **capture-less** lambdas can be implicitly converted to a function pointer. If lambda requires capturing some variables, it *can not be cast* to a function pointer.

Example:

```
int a;
auto f1 = []() {return 123; };
auto f2 = [&a]() {return a; };

int (*f1_ptr)() = f1;    // OK
// int (*f2_ptr)() = f2; // Compile time Error
```

Starting from C++14, there is an extra feature for Lambdas **"init capture"**. This feature allows performing arbitrary declaration of *closure* data members:

```
auto interpolate =
[min = toFloat(0), max = toFloat(255)]
(int value)-> float
{
  return (value - min) / (max - min);
};
```

Next, starting from C++14, you may not want to specify types of arguments in a way that they are derived during compile-time. This form of lambda is called **generic lambdas**.

A *generic lambda* is a lambda expression with at least one placeholder type such as:

- `auto`
- `auto&`
- `const auto&`

That turns the function call operator of the generated class for lambda into a template call method.

Example:

```
auto test = [](const auto& x){std::cout << x;};
```

Starting from C++20, the same `auto` syntax can also be applied to generic functions. Both of those things are just shorthand for template methods.

Finally, starting from C++20, it's possible to be more explicit about the type of such `template operator()` generation, and we can define what is called **lambda templates**. Example:

```
auto multiply = []<class T>(T a, T b)
{
  return a*b;
};
```

Move Semantics

The move semantics have been born due to realizing at some moment in C++ design that copy operation (especially deep copy) and move ownership of data are in fact *two different operations*.

Examples where move semantics will help:

- Your function returns a container or another object with the state in the heap. The function returns this container by value from a function call.
- You implement a container, and internally, your container has a dynamic array that changes its size once it's filled or based on another criterion. The internal memory storage of the container, when resizing happens, almost always consists of several steps: allocate new memory, copy objects to the new memory, destroy old objects, and release old memory. With **move** semantics, the authors of containers can leverage the move mechanism and remove unnecessary copying.
- Swapping of two objects in practice may happen through using an intermediate variable *tmp*. But if you swap the contents of two objects where objects have a state in the heap, then you can easily benefit from move semantics to avoid two copies.

Move semantics is a language construction to represent such action, but adding it to the language increases the complexity of references. Move semantics were one of the main innovations of C++11. The move is useful for objects that store part of their state in a heap. Moving is never slower than copying and is usually faster. In addition, for some objects, there are only moving semantics, e.g., an executable thread.

The move brings the object from which the move is performed *from* to a valid but unspecified state. What is less well known is that objects can be reused after movement; however, in practice, it does not happen frequently.

One more time - reusing an object after "moving from it" is legal and valid because the object's content should be in a valid state after the moving operation if the content has been moved from this object.

It doesn't make sense to implement a **move** functionality for a class/struct if:

(1) Its speed is worse or the same as copying

(2) This operation is not on the critical execution path in the program

(3) The object does not have any state in the heap

(4) You shouldn't use `std:: swap` to implement the move. It doesn't make any sense

(5) It makes no sense to implement a move if it's the same as a copy

The move semantics essentially mean that you want to grab the content of the object. The object from which you want to grab the content (or move from) should have an *rvalue reference* and grammatically written as `A&& a = (A&&)b;`.

As it has been mentioned in glossary, there are two sources for rvalue reference:

- A reference to an object that will soon be deleted (xvalue expression), which is typically an unnamed object.

- Explicitly and unconditionally cast the reference to the object through `std::move`. This is a wrapper over explicit casting to rvalue reference.

Why was it designed so special?

It is unsafe to move from the LValue reference. LValue reference is a live object, and it's very *unlikely* that you want to grab its content.

In contrast, a reference to an object that will soon be deleted is something from which we can grab the content because nobody will know whether we had the content or not. And so, e.g., unnamed objects are very nice candidates for moving. This is a reason why language is constructed in this way.

Important rule: Move from LValues is never executed. It is copied.

An RValue reference is denoted as `T&&`. You can overload any function or method in the class for the situation when it obtains as input LValue reference (`T&`) or RValue reference (`T&&`). As a rule of thumb *RValue references* are more important for overload resolution and therefore have higher priority.

The C++ compiler will try all possible functions. For `LValue` objects, calling an overload function that obtains an RValue reference is illegal, and this consideration is discarded. If you call the function `f()` with passing the xvalue (the temporary object that soon will be destroyed), then in principle, both variants can be called:

- `void f(T&&)`
- `void f(T&)`

But as it has been said, the construction `void f(T&&)` takes precedence over `void f(T&)` if both can be called. If function `void g(T&&)` has only a signature for rvalue reference, and you call it with an LValue reference, then it leads to a compile-time error.

Using const rvalue reference forms a valid C++ code; however, in practice, it is useless:

```
class A {};
void f(const A&&) {}
```

Some Details about Move Semantics

1. Commonly, it's a good idea to declare a function, constructors, and operators that use move semantics as `noexcept`. It's not a requirement of the C++ language. However, it's very often in practice because `noexcept` gives more compile-time optimization opportunities in this case. Therefore, move operations need not be `noexcept`, but it's preferable.

2. C++11 considers `noexcept` calling any destructor or memory allocation.

3. Move self to self - usually not checked because this is not normal behavior. C++11 defines that behavior is undefined in case it moves to itself.

The First Scary Thing of C++11: Handling RValue Refs with std::move

The move constructors and move equal operators for your classes should be written carefully.

```
#include <iostream>
#include <utility>

class M {
public:
    M()
    {
        std::cout << "M::M()" << std::endl;
    }

    M(const M&)
    {
        std::cout << "M::M(const M&)" << std::endl;
    }

    M(M&&)
    {
        std::cout << "M::M(M&&)" << std::endl;
    }
};

class MyClass {
public:
    MyClass(){}

    MyClass(MyClass&& theM)
    :
        // m (theM.m)          // Copy - not move
        m(std::move(theM.m)) // Move !
    {
    }

    M m;
};

int main()
{
    MyClass my;
    MyClass my2(std::move(my));

    return 0;
}
```

The fact is that `M&& theM` is an RValue reference, but in the context of the function body, `T&& arg` can be observed as a variable, and it makes `theM` an LValue. So you should be careful when you write your initializer list and call the base class. This is the most confusing thing about C++11 due to Scott Meyers.

Therefore, for example, in the initialization list in your copy constructor, you need to write in the initialization list `Base(std::move(rhs))`.

The `std::move` is a useful utility function that turns any LValue expression into an RValue reference; `std::move` could be called `rvalue_cast`.

- In fact `std::move()` does not move anything.
- Simplified implementation:

```
template <typename T>
T&& move(T& x) noexcept
{
  return static_cast<T&&>(x);
}
```

- More advanced implementation:

```
template<typename T>
typename std::remove_reference<T>::type&&
move(T& obj) noexcept
{
  using RetType = typename std::remove_reference<T>::type&&
  return static_cast<RetType>(obj);
}
```

To implement a move operation in your classes, you should typically call std::move available from .

The Second Scary Thing of C++11: Universal references.

You can create a special construction that uses `T&& param,` and it is valid only for template constructions in the following form:

```
template <class T>
void f(T&& arg)
{}
```

This construction is processed by special rules. For details, please check reference collapsing section of this note. But currently, we need two rules:

- arg is Lvalue => `void f<T&> (T&&,& arg)=>void f<T&> (T& arg)`
- arg is Rvalue => `void f<T&&> (T&&,&& arg)=>void f<T&&>(T&& arg)`.

That construction has an informal name **universal reference** (the term from Skott Mayers), and it binds everything (lvalues and rvalues):

```
template <class T>
void f(T&& arg)
{}
```

The universal reference is not the official term. Scott Myers coined it only for informal discussions. Even though everybody (including language designers) realizes that it's a very useful concept.

The `auto` type deduction works in the same way as the template type deduction and has the same reference collapsing rules:

```
int f();
int x;
auto&& z1 = f();  // z1 type is int&&
auto&& z2 = x;    // z2 type is int&
```

While using universal references, the situation is a bit exotic. One template function can actually create two different functions, one for **lvalue** and one for **rvalue**.

In the context of moving, two operations are typically needed from the C++ Library:

- std::move - unconditional RValue reference cast. The usage of such functionality is named **move cast**. We have already seen it.
- std::forward - conditional cast of universal references to the needed reference type. Copies LValue arguments, moves RValue arguments. It's applicable only to function templates. Its utility is preserving arguments LValueness/RValueness/constness when forwarding them. The usage of such functionality is named **perfect forwarding**. The std::forward is designed to be used with universal references.

Example of using `std::forward`:

```
class Point {
public:
  template<typename T1, typename T2>
  void setNameAndCoords(T1&& n, T2&& c)
  {
    name = std::forward<T1>(n);
    coordinates = std::forward<T2>(c);
  }

private:
  std::string name;
  std::vector<int> coord;
};
```

The simplified implementation of std::forward based on using reference collapsing rules:

```
template <typename T>
T&& forward(T&& a) noexcept {
    return static_cast<T&&>(a);
}
```

Virtual Functions and Polymorphism in C++

General Rules from C++03

To get runtime polymorphic behavior, the member functions must be `virtual`, and objects must be manipulated through pointers or references. For a function to behave **virtually**, its declaration in a derived class must have the same signature as it does in the base class.

> The only one exception for virtual function declaration is allowed, and it is about the return type. The derived class version of a "*virtual function*" may return a pointer or a reference to a more specialized type than the base. The technical term used in relation to these return types is *covariance*. [4, p.576].

```
#include <iostream>

class B
{};
class D: public B
{};

class BB {
public:
    virtual B* f() {
        std::cout << "BB::f" << '\n';
        return 0;
    }
};

class DD:public BB {
public:
    D* f() override
    {
        std::cout << "DD::f" << '\n';
        return 0;
    }
};

int main()
{
    DD d;
    BB& b = d;
    b.f();

    return 0;
}
```

When you're calling a method using the scope resolution `operator::` this ensures that the virtual mechanism built in the language **is not used at all**. This call will be resolved at compile time only.

We said that objects must be manipulated through pointers or references to invoke a virtual mechanism. In the case of manipulating an object directly (rather than through a pointer or references) - its exact type is known to the compiler at compile time. In that case, runtime polymorphism is not needed, and the call will be done without using virtual pointers (vptrs).

A virtual function invoked from a constructor or a destructor reflects that the object is partially constructed. It's easier to remember it in the following way - *during constructor or destructor invocation, the virtual mechanism is not used at all*.

```
#include <iostream>

struct A
{
    virtual void f(){std::cout << "A::f()\n";}
};

struct B: A
{
    virtual void f(){std::cout << "B::f()\n";}
};

int main()
{
    B obj;
    obj.f();    // B::f()
    obj.A::f(); // A::f()

    return 0;
}
```

Override Specification (from C++11)

The **override specification** (cpp reference details) of a virtual function guarantees that you have not made any mistakes in the function signatures at the time of writing. It safeguards you and your team from forgetting to use the virtual function signature in the base or derived class at the same time.

The use of `override` is optional, but being explicit allows the compiler to catch mistakes, such as misspellings of function names or slight differences between the types of virtual functions. Let's describe it in detail.

It can be a situation where the function is intended to override something, but due to a mistake, it is a new virtual function. In this case, using `override` will remove this problem.

Better not to repeat `virtual` in a derived class, but if you want to be explicit, use `override`.

> B. Stroustrup: "It is illogical that virtual is a prefix and override is a suffix. This is part of the price we have paid for compatibility and stability over the decades. Curiously, override is not a keyword; it is what is called a contextual keyword. E.g. it can be used as an identifier."

```
struct Base
{
    Base(){puts("Base()");}
    Base(const Base&){puts("Base(const Base&)");}
    // declare functions in a base class that can be redefined in each derived
class
    virtual void f(){}

    // declare functions in a base class that can be redefined in each derived
class
    virtual void g(){}
};

struct A: Base
{
    A(){puts(__func__);}
    A(const A&a): Base(a) {puts("A(const A&)");}
    // By default, a function that overrides a virtual function itself becomes
virtual
    virtual void f(){}
    // By default, a function that overrides a virtual function itself becomes
virtual
    void g(){}
};

struct B: Base
{
    B(){puts(__func__);};
    B(const B& b) : Base(b) {puts("B(const B&)");}
   // 1) Better to not repeat virtual in a derived class
   // 2) If you want to be explicit, use "override."
   void f() override final {}
   // Final provides prevent further overriding
   void g() override final {}

   int override = 7;
};
```

Final Specification

The **final specification** prevents a member function from being overridden in a derived class. This could be because you want to limit how a derived class can modify the behavior of the class interface. There is no contradiction in combining override and final. This states that you disallow any further overrides of the function you are overriding. [5, p.578].

It is also possible to specify an entire class as **final**. That will enforce that no further derivation from the final class is possible.

Example:

```
```

```
class A final
{};

// Compile time error
// class B: public A {};
```

A function's access specifier determines whether you can call that function. It plays no role whatsoever, though, in determining whether you can override it. The consequence is that you can override a private virtual function of a given base class.

Example:

```
struct Base
{
public:
    Base() { puts("Base()"); }
    Base(const Base&) { puts("Base(const Base&)"); }
private:
    virtual void f() {}
};

struct A: Base
{
public:
    A() { puts(__func__); }
    A(const A& a) : Base(a) { puts("A(const A&)"); }
    void f() override final {}
};
```

Connection of Virtual Function with Default Values

If you call the function through a base pointer, you will always get the default argument value from the base class version of the function.

Any default argument values in derived class versions of the function will have no effect.

Miscellaneous Features of C++11

1. emplace_back

The `emplace_back` construct object is directly in place in the memory of the container.

Documentation: cpp reference details.

2. vector::shrink_to_fit

`vector::shrink_to_fit` method requests the removal of unused capacity from the reserved memory of `std::vector` that encapsulates the dynamic linear array.

Documentation: cpp reference details.

3. noexcept function specification

The `noexcept` specification is like the `throw()` specification for functions from C++98/03, but it allows for more optimization. Also, `throw()` is an explicit exception specification.

> The exception specification has been deprecated in C++11 and removed from C++17.

Example:

```
void g1(int) throw() {}  // Valid only for C++98/03/11/14
void g2(int) noexcept {} // Valid from C++11
```

Documentation: cpp reference details.

4. static_assert expression

The syntax for `static_assert` is the following:

- `static_assert(expr)`
- `static_assert(expr, message)`

This is a special declaration that results in a compilation error if the constant expression `expr` is evaluated as false. The `static_assert` is valid "anywhere":

- Global and namespace scope
- Class scope
- Function and block scope

Example:

```
static_assert(sizeof(void*) == sizeof(int),
              "Pointers and integers are different sizes");
```

Documentation: cpp reference details.

If the static_assert keyword is not supported by your compiler or you need custom behavior for compile-time errors, you can implement your own version as follows:

```
// Example of implementing own static assert with templates means
template <bool> struct static_assert;
template <> struct static_assert<true> {};
```

5. alignas operator

The `alignas` operator from (C++11) enforces alignment. The `alignas` allocates an object with the requirement to allocate it with an alignment suitable for a specific type. Example of usage:

```
alignas(int) char buff[1024];
```

Documentation: cpp reference details.

6. alignof operator

The `alignof(x)` is the operator built into the language that returns the alignment of a type in memory in bytes.

Example:

```
  std::cout << alignof(char);
  // print alignment for char's
```

Documentation: cpp reference details.

7. Default Member Initializers (C++11)

When a class data member is defined, we can supply a default initializer called a default member initializer. The default value is used whenever a constructor doesn't provide a value.

This simplifies the code and helps us avoid accidentally leaving a member uninitialized.

```
class X {
  int i = 4;
  int j {5};
};
```

Documentation: cpp reference details.

8. User-Defined Literals (UDL)

Unsurprisingly, literals with user-defined suffixes are called user-defined literals or UDL.

```
long double operator "" _w(long double);
int main() {
    double z = 1.2w; // calls operator "" _w(1.2L)
    return 0;
}
```

Documentation: cpp reference details.

9. noreturn Attribute

Placing `[[noreturn]]` at the start of a function declaration indicates that the function is not expected to return.

Example:

```
[[noreturn]] void exit(int);
```

Documentation: cpp reference details.

10. Anonymous Unions

Anonymous unions may help you to use stack space more effectively.

```
  union {
     float f;
     uint32_t d;
  };
  f = 3.14f;
```

Documentation: cpp reference details.

11. Type Alias

Starting from C++11, there are "*Alias Templates*" also known as "*Smart Typedefs*". With this form of new typedefs, typedef declarations can now be used for "partially bound" templates:

```
// Usual namespace alias from C++98. Another name for std
namespace AA = std;

// Usual typedef: Another name for AA::vector<int>
typedef AA::vector<int> VecInt1;

// Alias for binding some types: typedef analog from C++11
using VecInt2 = AA::vector<int>;

// MyAllocVec is an alias template: not implementable via typedefs
template <class T>
using MyAllocVec = std::vector<T, MyAllocator>;
```

Remarks about Alias Templates:

- They cannot be partially specialized
- They can be used wherever `typedef` is used.

Documentation: cpp reference details.

12. Static Variables are Always Initialized Thread Safe

Interestingly, that `static` local variable in C++11 guarantees thread-safe initialization. This was not the case in C++03.

13. Delegating Constructors

A constructor can call another constructor in the initialization list. In that way, initialization is delegated to another constructor. The delegates can also delegate construction to another constructor.

Example:

```
class MyArea
{
public:
  explicit MyArea(double w_and_h)
  : MyArea(w_and_h, w_and_h) {}

  MyArea(double theW, double theH)
  : w(theW), h(theH)
  {}

private:
  double w;
  double h;
};
```

There is one extra thing regarding delegating constructors. Once you have decided to delegate construction work to another constructor, you **can not** initialize any member in the initializer list of constructors.

If you add member initialization to the previous code snippet, it will lead to a compile error:

```
// Compiler Time Error:
class MyArea
{
public:
  explicit MyArea(double w_and_h):
  MyArea(w_and_h, w_and_h), h(1.0) {}
//...
};
```

Documentation: cpp reference.

14. Inheriting Constructors

Inherited constructors mean generating new implicit constructors that are called the base class versions. That brings the ability to use `using` declarations with base class constructors.

```
class B {
public:
  explicit B(int);
  void f(int);
};

class D: public B {
public:
  // Bring f() into the scope of class D: valid from C++98
  using Base::f;

  // Implicit declaration: D::D(int a):B(a){} (valid from C++11 only)
  using Base::Base;

  void f();

  D(int x, int y);
};

Derived d1(1);
Derived d2(2, 3);
```

Documentation: cpp reference details.

15. Destructors are Implicitly noexcept

Starting with C++11, destructors are normally implicitly `noexcept`. Even if you define a destructor without a `noexcept` specification, the compiler will normally add one implicitly [4,p.635].

16. Fixed Width Integer Types

Since C++11, various fixed integer types like `std::int64_t` are a part of the library and are available from `<cstdint>`.

Documentation: cpp reference details

17. Concurrency Support

For the first time, concurrency support has been introduced into the language for C++11. Before C++11, the language was not aware of multithreading aspects at all at the level of the standard library.

- std::thread. Independent execution by real OS threads. Thread detachment is supported when no thread joining is needed.

- std::async(). Request asynchronous execution of a function. Async support launch deferred policy to assist in debugging asynchronous code in a single thread.
- std::future<ret_type>. A token representing the function result and encapsulating the thrown exception.
- Mutexes:
    - std::mutex
    - std::timed_mutex
    - std::recursive_mutex
    - std::recursive_timed_mutex
    - std::shared_timed_mutex
- Condition Variables (std::condition_variable). To allow threads to communicate about changes in shared data via the ability to notify one/all waiting threads.
- thread_local data with static storage duration (with internal `static` or external `extern`) for thread-specific data.
- std::atomic atomic types (e.g., `std::atomic<int>`) with memory ordering options.
- std::call_once for thread-safe only one-time function (or callable object) invocation, even if called from several threads.
- Thread-safe initialization guarantees of objects of static storage duration.
- There are some library thread safety guarantees (e.g., for std::cin/std::cout, containers). Please check ([9], link).

18. Explicit Conversion Functions

In C++98/03, the `explicit` keyword is used only for the constructor to prevent implicit type conversion during a call to a constructor.

Starting from C++11, the `explicit` keyword is now applicable to type conversion operators. It will prohibit using type conversion implicitly, only explicitly.

Example:

```
class MyClass {
public:
    //...
    // C++11 only
    explicit operator std::string() { return ""; }
    //...
};

int main()
{
    MyClass a;
    std::string s1 = static_cast<std::string>(a);    // OK
    std::string s2 = (std::string)(a);               // OK
```

```
    // std::string s3 = a;                              // Compile-Time Error
    return 0;
}
```

It is worthwhile to refresh that in C++, when using custom conversions via constructors without `explicit` keywords or defined `type conversions`, only one level of implicit conversions is allowed. It has been written in [1], but it's only one place where such information has been mentioned. Documentation: cpp reference details.

19. Current Exception. Internal Details.

The construction in the standard library is devoted to the concept of the current exception object called "*Exception Pointer*". Starting from C++11, it's possible to get an object via std::current_exception and std::exception_ptr.

20. The Trailing Return Type For Functions

Documentation: cpp reference about trailiing return type.

```
auto sumA(int a, int b) -> int {
    return a + b;
}

auto sumB(int a, int b) -> decltype(a+b) {
    return a + b;
}
```

21. Return Type Deduction

Documentation: cpp reference about placeholder type specifier.

```
// Return type deduction in templates: auto deduces to value type
template <class T>
auto sum1(T a, T b)
{
    return a + b;
}

// Return type deduction in templates: auto deduces to the exact type of return
expression
template <class T>
decltype(auto) sum2(T a, T b)
{
    return a + b;
}
```

22. Built-in Array type

Starting from C++11, the new standard template type std::array has been added to the standard library. Essentially it is a container to hold an array of elements of a fixed size.

Miscellaneous Features of C++14

1. Deprecated Attribute

This indicates that the use of the name or entity declared with this attribute is allowed but discouraged for some reason.

Compilers typically issue warnings during the usage of this function. For example:

```
[[deprecated]] void func(int);
```

Documentation: cpp reference details about deprecated.

2. Return Type Deduction

For usual functions, it is now possible to deduce the return type from its return statements similar to C++11 templates.

```
auto f() {
   return 42;
}
```

Documentation: cpp reference details about auto.

3. Binary Literals

In C and C++, for integers of different sizes and types, you can use decimal literals (`123`), hexadecimal literals (`0xFF, 0Xff`), and octal literals (`071`).

Starting from C++14, there is support for binary literals. You can write a binary integer literal as a sequence of binary digits (0 or 1) prefixed by either `0b` or `0B`.

```
unsigned char a = 0b00110011;
```

Documentation: cpp reference details.

4. Variable Templates

Example:

```
template<class T>
inline constexpr T pi = T(3.14159)
```

Documentation: cpp reference details.

5. Delimiter Inside Numeric Literals

You can use the single quote character to separate digits for readability in integer literals. Example:

```
long long d {10'000'000LL};
```

C++ does not impose any restrictions on how to group the digits. Documentation: cpp reference details.

6. std::make_unique

C++14 has changed the recommended way to create a `unique_ptr`. Now, the recommended way is to employ the `std::make_unique<T>()`. Example:

```
#include <memory>
struct Vec {
    int x, y;
};
std::unique_ptr<Vec> v = std::make_unique<Vec>();
```

Documentation: cpp reference details.

Miscellaneous Features of C++17

1. Structured Binding

```
struct Entry {
   string name;
   int value;
};
auto [n,v] = read_entry(is)
```

The `auto [n,v]` declares two local variables n and v with their types deduced from `read_entry()` return type. This mechanism for giving local names to members of a class object is named a structured binding. You can use this mechanism for an arbitrarily large number of variables.

Documentation: cpp reference details.

2. Deduce Template Parameters from Ctor. Arguments

```
template <class T>
class A {
public:
    A(T arg)
    {
      (void)arg;
    }
};
```

Starting from C++17, it's possible to just write:

```
A a(123); // valid from C++17
```

instead of

```
A<int> a(123); // Correct syntax in C++98/03/11/14 (and only one correct)
```

Before C++17, such a feature was supported only for template functions, not for template classes. That feature is called *Class Template Argument Deduction*.

Documentation: cpp reference details.

3. Compile Time if

Only the selected branch is instantiated by using some compile-time expression. This solution offers optimal performance and the locality of the optimization.

```
#include <type_traits>
template<typename T> void update(T& target)
{
 if constexpr(std::is_pod<T>::value)
    simple_and_fast(target); // for "plain old data"
 else
    slow_and_safe(target);   // ...
}

int main() {
   return 0;
}
```

Documentation: cpp reference details.

4. __has_include Macro

The new macro `__has_include` is used to test that the header file is presented. Example of usage:

```
#if __has_include(<optional>)
#endif
```

Documentation: cpp reference details about __has_include, C++2020 standard, p.446, Section 15.2

5. std::byte

C++17 introduces the `byte` type to work directly with bytes. The `std:: byte` is defined in `<cstddef>`.

Documentation: cpp reference details.

6. Fallthrough Attribute

In C and C++ language inside `switch` construction, if you do not explicitly use `break` instruction, then the instruction flow once it finds a suitable `case` will fall through the next `case` statement(s) without checking the predicate for `case`.

The C++17 added a language feature to signal the compiler and the person reading your code that you are *intentionally* using a fall-through.

Example:

```
switch (number)
{
case 1:
 ;
 [[fallthrough]];
case 29:
case 78:
 ;
 break;
//...
}
```

Documentation: cpp reference details.

7. Initialization Statements in if/switch/for

The syntax for defining local variables in `if` statement was common. And in C++17, a specialized syntax for it was added in several constructions via using syntax similar to `for`, but now it's available to perform initialization in:

- `if`
- `switch`
- range-based-for loop statements:

Examples:

- Initialization in `if` statement:

```
if (auto my=2; my >= 1 || my <= 2) {
  ;
}
```

- Initialization in `switch` statement:

```
switch (initialization; condition)
{ ... }
```

- Initialization in range-based `for` loop statement:

```
for ([initialization;] range_declaration : range_expression)
// loop statement
```

```
#include <iostream>

int main()
{
    int v[] = {1,2,3};
    for (int init = 123; auto i : v)
        std::cout << i;
}
```

8. std::optional

The `std::optional` is a class template for conceptually storing an object. The optional object optionally contains a value.

To check that `std::optional` has value in it, you have *three ways*:

- Convert the `std::optional` object to a bool.
- Call the `has_value()` member function.
- Compare the `std::optional` object to `std::nullopt`.

Example:

```
#include <iostream>
#include <optional>
```

```
int main()
{
    std::optional<int> o1;
    std::optional<int> o2 = 1;

    std::cout << *o1 << ' ' << *o2 << "\n";
    std::cout << (o1.has_value() ? "YES" : "NO") << ' '
              << (o2.has_value() ? "YES" : "NO") << "\n";

    std::cout << ( (bool)o1 ? "YES" : "NO") << ' '
              << ( (bool)o2 ? "YES" : "NO") << "\n";

    std::cout << ( o1 != std::nullopt ? "YES" : "NO") << ' '
              << ( o2 != std::nullopt ? "YES" : "NO") << "\n";
}
```

Documentation: cpp reference details. Feature test macro: __cpp_lib_optional.

9. std::string_view

The type `std::string_view` has been introduced in C++17. It can be used instead of `const std::string&` for input string parameters.

Initializing or copying a `string_view` is very cheap, but use it only when you understand what you are doing! It is the program logic writer's responsibility to ensure that `std::string_view` refers to a valid character array.

Similar to `std::string_view` there is a `std::span<const T>` that can be used instead of `const std::vector<T>&`.

The `std::span` and `std::string_view` provide a way to get rid of making an extra copy of data via using "light views".

Example:

```
#include <iostream>
#include <span>
#include <string_view>
#include <string>
#include <vector>

int main()
{
    const char* s = "my string";
    std::string_view sv = s;
    std::string str = s;

    // MATCHES
    std::cout << "std::string_view and raw string: "
              << (s == sv.data() ? "MATCHES" : "NOT MATCHES")
              << "\n";
```

```
    // DOES NOT MATCH
    std::cout << "std::string and new string: "
              << (str.data() == sv.data() ? "MATCHES" : "NOT MATCHES")
              << "\n";

    std::vector<int> arr = {1,2,3};
    std::span<int> my_span = arr;

    // MATCHES
    std::cout << "std::span and std::vector: "
              << (my_span.data() == arr.data() ? "MATCHES" : "NOT MATCHES")
              << "\n";

    return 0;
}
```

Documentation: cpp reference details about string_view, and cpp reference details about span

10. Inline Variables

Static variables with inline initialization start to be supported since C++17.

```
// C++17 simplified static variables declaration
class Objects {
    static inline size_t s_object_count {};
};
```

Before inline variables, it was possible to use static variables, but the *burden of adding static variables into compilable* and, finally, the linkable binary was under your responsibility.

```
// Somewhere header file
class ObjectsOld {
    static size_t s_object_count;
};

// Somewhere in some cpp unit
size_t ObjectsOld::s_object_count;
```

Documentation: cpp reference details about inline specifier.

> If you want to precisely understand exactly how the toolchain handles inline variables (i.e., in which object file this variable is defined) in situations when projects consist of several compiled libraries and binaries, then please double-check with Toolchain documentation. If your project is built with the same tools, it should not induce any problems.

It may be worthwhile to consider the following example in addition due to some inherited subtleties:

```cpp
#include <iostream>

class MyClass
{
public:
    static const int value_i = 1;
    static inline const float value_fa = 2.0;

//  COMPILE-TIME ERROR
//    static /*inline*/ const float value_fb = 3.0;
};

//  COMPILE-TIME ERROR
// const float MyClass::value_fa;

// NOT COMPILE-TIME ERROR AND REQUIRED DUE TO ORIGINAL C++ DESIGN INTENT
const int MyClass::value_i;

int main()
{
    std::cout << MyClass::value_i << std::endl;
    std::cout << MyClass::value_fa << std::endl;
}
```

The initialization of `value_i` looks like `static inline` C++17 initialization, however, in fact, this is the old construction that has been in the language since C++98.

The initialization construction for `value_i` is named as the initialization of integral static constants directly in the class.

According to [1], 10.4.6.2: "You can initialize a class member that is a static constant of integral type. If and only if you do so, you must declare the member somewhere once...The member shall still be defined in a namespace scope if it is used in the program and the namespace scope definition shall not contain an initializer..."

The C++ Standard - ANSI ISO IEC 14882 2003 describes this construction in the section "9.4.2 Static data members, Subparagraph 4":

> If a static data member is of const integral or const enumeration type, its declaration in the class definition can specify a constant-initializer, which shall be an integral constant expression (5.19). In that case, the member can appear in integral constant expressions.

In real compilers, in fact, the definition `const int MyClass::value_i;` is optional in the sense that code with and without this line can be compiled since GCC 4.5.3 and MSVC 2012 and later.

11. The Exception Specification has been Removed

The exception specification has been deprecated in C++11 and removed from C++17.

Details: Core Working Group about removing Deprecated Exception Specifications from C++17

12. Built-in Function for Determining the Size of the Array

Documentation: https://en.cppreference.com/w/cpp/iterator/size

```
#include <iostream>
#include <array>

int main()
{
    int a[3];
    std::cout
    << "Array size in items: "
    << std::size(a);

    return 0;
}
```

13. Short Syntax for Nested Namespaces

A nested namespace is essentially the namespace inside another namespace. Example of code valid for C++98:

```
// Valid from C++98
namespace MyLib {
  namespace Module {
    namespace SubModule {
      class Mylass{};
    }
  }
}

int main()
{
    MyLib::Module::SubModule::Mylass obj;
    return 0;
}
```

Starting from C++17 to express the same concept there is a short syntax:

```
// Valid from C++17
namespace MyLib::Module::SubModule {
    class Mylass {};
}

int main()
{
    MyLib::Module::SubModule::Mylass obj;
```

```
    return 0;
}
```

Documentation: https://en.cppreference.com/w/cpp/language/namespace

14. Filesystem

The filesystem support is defined in filesystem. It allows you to write portable code to work with a file system. Functionality includes: querying whether the path is a directory or a file, iterating over the directory, manipulating paths, and retrieving information about files, etc.

Feature-test macro: __cpp_lib_filesystem

Documentation: cpp reference filesystem.

Miscellaneous Features of C++20

1. no_unique_address Attribute

The `[[no_unique_address]]` attribute allows empty non-static data members to share space with another sub-object of a different type.

Example based on materials from CPP reference:

```
#include <iostream>
struct Empty  {};

struct X {
    int i;
    [[no_unique_address]] Empty e;
};

int main() {
    X obj;
    std::cout << ( (void*)&obj.i == (void*)&obj.e );
    return 0;
}
```

Another use case is utilizing this feature in situations where we have a pair of objects, but one of the objects has no data members. The `no_unique_address` attribute can be applied to avoid wasting a single 1 byte, thereby preventing wasting this 1 byte (and extra additional padding bytes) for storing a single pair:

```
#include <iostream>
#include <stdint.h>

struct Empty {};
```

```
#ifdef _MSC_VER
    #define no_uniq_address_attr [[msvc::no_unique_address]]
#else
    #define no_uniq_address_attr [[no_unique_address]]
#endif

template <typename TypeA, typename TypeB>
struct CompressedPair
{
    no_uniq_address_attr TypeA a;
    no_uniq_address_attr TypeB b;
};

template <typename TypeA, typename TypeB>
struct UnCompressedPair
{
    TypeA a;
    TypeB b;
};

int main()
{
#if __has_cpp_attribute(no_unique_address)
    std::cout << " cpp attribute [[no_unique_address]] is supported [OK]\n";
#else
    std::cout << " cpp attribute [[no_unique_address]] is not supported [BAD]\n";
#endif

#if __has_cpp_attribute(msvc::no_unique_address)
    std::cout << " cpp attribute [[msvc::no_unique_address]] is supported [OK]\n";
#else
    std::cout << " cpp attribute [[msvc::no_unique_address]] is not supported
[BAD]\n";
#endif

    std::cout << "sizeof(Empty): " << sizeof(Empty) << " bytes\n";
    CompressedPair<int64_t, Empty> pair_1;
    std::cout << "sizeof(CompressedPair<int64_t, Empty>): " << sizeof(pair_1) << "
bytes\n";
    UnCompressedPair<int64_t, Empty> pair_2;
    std::cout << "sizeof(UnCompressedPair<int64_t, Empty>): " << sizeof(pair_2) <<
" bytes\n";

    return 0;
}
```

Documentation: cpp reference details.

2. Spaceship Operator

The three-way comparison operator denoted `<=>` is a new comparison operator. It has the informal name - **spaceship operator**. The term "*spaceship operator*" was coined by Randal L. Schwartz because it reminded

him of the spaceship in the 1970s text-based strategy video game Star Trek. [4,p.100].

```
std::strong_ordering operator <=> (const Y&, const Y&) {}
```

```
std::partial_ordering operator <=> (const Y&, const Y&) {}
```

```
std::weak_ordering operator <=> (const Y&, const Y&) {}
```

After defining what is called a *spaceship operator* the compiler generates based on it various comparison operators: `<`, `>`, `<=`, `>=` automatically.

The `<=>` operator for your class can return one of the following types:

- std::strong_ordering. It's an enumeration type of available values:
    - std::strong_ordering::less
    - std::strong_ordering::greater
    - std::strong_ordering::equal.

    In a mathematical sense, it should be used when the relation is ordered.
- std::partial_ordering. It's an enumeration type of available values:
    - std::strong_ordering::less
    - std::strong_ordering::greater
    - std::strong_ordering::equivalent
    - std::strong_ordering::unordered

    In a mathematical sense, it should be used when the relation is partially ordered.
- std::weak_ordering. It's an enumeration type of available values:
    - std::strong_ordering::less
    - std::strong_ordering::greater
    - std::strong_ordering::equivalent

Example:

```
struct Y {
    int y;

    std::strong_ordering operator <=> (const Y& rhs) const {
        return y <=> rhs.y;
    }
};
```

Documentation: cpp reference details.

3. likely and unlikely Attribute

The `[[likely]]` and `[[unlikely]]` attributes are mainly applicable for `case` branches in the switch statement and `if` and `else` branches in a conditional statement.

They give a compiler a hint on optimizing certain branches. Example:

```
switch(value) {
  case 1:
    break;
  [[likely]] case 2:
    break;
}
```

```
if (a > 0) [[likely]]
  return a;
else [[unlikely]]
  return 0;
```

Documentation: cpp reference details.

> It can be a potential debate whether it is good or not to throw a ball to the software engineering side from specialized optimization tools and the compiler itself, but in any case, such functionality now exists.

4. std::format()

The `std::format()` replaces sprintf and ostringstream.

After importing C++20 `<format>` module or including `<format>` header files, you can use `std::format` in the following way:

```
std::cout <<
  std::format("diameter required for {} is {:.2f}.\n", x, y);
```

A slightly simplified general form of the format specifiers is the following:

`[[fill]align][sign][#][0][width][.precision][type]`.

For full information about the format specification, look into Standard format specification

Documentation: cpp reference details.

> Unfortunately, due to compiler support C++20 information the `std::format` is supported only from GCC-13, Clang 14, MSCVC 19.29. To use this feature, you need to use modern versions of these toolchains.

5. source_location::current()

Compile time information about the current source file `source_location::current()` defined in `<source_location>`.

Behaves in an implementation-defined way, but essentially it is analogous to predefined macros `__LINE__` and `__FILE__`.

The `source_location::current()` contains information about `line`, `column`, `file_name`, `function_name`.

Feature-test macro:

__cpp_lib_source_location

Documentation:

cpp reference details.

6. nodiscard(reason) attribute

Attribute `[[nodiscard(reason)]]` applied for function and described the consequence of discarding the return object from the function call.

The compiler is encouraged to issue a warning in case of discarding the return value.

Example:

```
[[nodiscard("DO NOT IGNORE")]] int sum(int x, int y)
{ return x + y; }
```

Documentation: cpp reference details.

7. Only One Signed Integer Representation

Starting from `C++20`, there is only one signed integer representation, and it's **twos-complement-notation**.

Documentation: cpp reference details about fundamental types.

8. Right Shift is Arithmetic Right Shift

From C++20, the right shift on signed integral types is an **arithmetic right shift**, which performs sign-extension. Before C++20, a formal right shift led to undefined behavior, but it is *not* undefined behavior starting from C++20.

Documentation: cpp reference details.

9. Abbreviated Function Templates

Starting from C++20, the template function can be written a bit more briefly. That syntax is called "*Abbreviated Function Template*".

Example:

```
// Classical syntax
template <typename T>
T sqr(T x) { return x * x; }

// Abbreviated Function Templates
auto sqrNew(auto x) { return x * x; }
```

If you want your template to instantiate functions where multiple parameters have the same type or related types, you still have to use the old syntax.

It is because every occurrence of `auto` or `auto&` in the function parameter list of an abbreviated function template introduces an implicit, unnamed **but** *new* template type parameter.

Documentation: cpp reference details.

10. Automatic generation of `!=` from `==`.

If you overload `operator ==` C++20 compiler automatically generates `operator !=`.

11. Designated initializers

The mechanism from C99, called "*named initialization*" was not previously available in C++03/11/17. Now it is available.

Example (from cpp reference):

```
struct A { int x; int y; int z; };
A b{.x = 1, .z = 2};
```

Documentation: cpp reference details

12. Feature Test Attribute

Inspired by some new features of C++20, for example `deprecated` attribute, we want to define a function:

```
[[deprecated]] void func(int);
```

However, we do not know about the support of `deprecated` attribute in the compiler itself.

Fortunately, such a check can now be done in compiler time via `__has_cpp_attribute`:

```
#if __has_cpp_attribute(deprecated)
  [[deprecated]] void func(int);
#endif
```

As it has been said in the standard:

It is expected that the availability of an attribute can be detected by any non-zero result.

Documentation:

- Draft C++2020 Standard, p.446, Section 15.2
- Feature Test in cppreference.

13. Feature Test Macro

In C++2020, the macros `__cpp_*` have been added to test language and library features under the name feature_test.

Support for this ability has been added from C++2020. Even though some features may have been supported in earlier versions, but the ability to check with macros has been added (officially) only since C++ 2020.

According to Draft C++2020 Standard, p.455, Section 15.11: *Future versions of this International Standard might replace the values of these macros with greater values.*

If you want to produce compatible code, use the following constructions (starting from C++2020). It can be used for feature tests:

```
#if __cpp_char8_t >= 201811L
///
#endif
```

In this conditional compilation block, the right-hand side has been taken from Table 19 of Draft C++2020 Standard, p.455, Section 15.11.

The list of various features since C++11 and the corresponding values for them can be obtained from Table feature_test. A potential confusion is that testing functionality has only been available since C++20.

14. std::span

The std::span class template allows you to refer to any contiguous sequence of T values and is analogous for std::string_view, but for arrays.

However, in addition to similarities std::span allows you to modify the underlying data even with `[]` operator.

15. Bit Manipulation

The Standard Library supports functions to work with bits of unsigned integral type. These bit manipulation routines are defined in bit header.

In fact, it also includes bitwise rotation operators typically available in the Assembly level, but which are not part of the C++ language as ROTL, ROTR.

Modules (from C++20)

The standard `#include` preprocessor directive helps organize the project, but it happens at a considerable cost. The desirability of modules due to B.Stroustroup was already well known in 1980.

Modules have come with C++20 to upgrade the understanding of header files. The reason to include it in the standards that B.Stroustrup mentioned in his talks is mainly about improving compilation time:

- Google reports *x2-4* improvements in compile-time
- Microsoft reports *x5-x50* times improvements in compile time

Result of applying modules:

- Reduce compilation time
- The order in which you import modules never matters

C++20 has introduced modules to form self-contained subcomponents of related functionality and called it a *module*.

Several critical conceptual things:

- A module can **export** any number of C++ entities (functions, constants, types, etc.)
- An exported entity by one module can be used in any source file that **imports** this module
- The combination of all entities that a module exports is called the **module interface**
- A module can **export entire other module**

Documentation dedicated for modules: cpp reference details

> In real life, compilers are not yet fully supported modules (at the moment of August 2022), so please be careful if you're going to use them. At least check that your toolchain endorses it from practice and from checking cpp reference details about compilers support if you are using one of the popular compilers.

Modules are replacements of header files and do not have any connection with C++ namespaces. Next, we will go example by example to open all features of *modules*.

Single Module Interface File/Module Unit

```
// math.cppm

// There is no consensus yet on what file extension to use for module files.

// *.cppm is c++ module file/module unit.

// 1. At the start of every module file is a module declaration
// 2. "export module ..." means that this is the module interface file
```

```
export module simple; // Declares the primary module interface unit


// 3. Within a module name, you may use dots (.) to concatenate together multiple
identifiers. Module names are the only names in C++ in which this is allowed.
// Example: export module simple.my.hello;

//-------------------------------------------------------
// 4. All import declarations must appear after the module declaration
//-------------------------------------------------------
// 5. To export an entity from a module, you simply add "export".

export auto square(const auto& x) { return x * x; }
export enum class Oddity { Even, Odd };
export auto getOddity(int x)
{
  return isOdd(x) ? Oddity::Odd : Oddity::Even;
}
// 6. Only module interface files may contain export declarations.
// 7. You can also export multiple entities all at once by grouping them into
export blocks.
export
{
  const double a{ 1.2 };
  const double b{ 1.5 };
}
//-------------------------------------------------------

// 8. Module Local function (not exported)
bool isOdd(int x) { return x % 2 != 0; }

// 9. Only entities that are exported by a module can be used in files that
"import" the module
```

Module Interface File With Implementation Inside It

First, you may declare export symbols and implement them inside the module interface file, as seen in the previous subsection.

```
export module simple;

export // The module's interface
{
 auto square(const auto& x);
}

// The implementation of the module's functions
auto square(const auto& x) { return x * x; }

// You can add export, but you do not have to. However, use export in impl. is
```

```
valid:
// export auto square(const auto& x) { return x * x; }
```

Module Interface File With Separate Implementation

The module interface file in this style includes the prototypes of all exported functions.

In a **module interface file**, all import declarations must appear after the module declaration only.

```
// mymodule.cppm – Interface test file
export module mymodule;
import <string>;
export std::string to_string(unsigned int i);
```

Definitions, along with any module-local entities, can be moved to one or more **module implementation files**.

```
// mymodule.cpp – Interface implementation file

// 1. Unlike module interface files, a module implementation file does not begin
with the export keyword.

// 2. You are not even allowed to repeat the export keyword here in front of the
definition of to_string

module mymodule; // declares a module implementation unit for named module 'A'
this should be presented in each mymodule implementation files

import <string>; // import declaration

std::string to_string(unsigned int i)
{
  return std::to_string(i);
}
```

Every module implementation file implicitly:

- Imports the module that it belongs to.
- Gains access to all declarations, even those that are not exported from the module interface file. [5,p.396]

What You Can not Define in a Module Implementation File

Unfortunately, some entities must always be defined in a **module interface file**:

1. The definitions of all exported templates,

2. A function definition with `auto` return type deduction. For auto return type deduction to work, the compiler needs the function definition to be part of the module interface.

Using Modules

Compiling a module interface creates a binary representation of all exported entities for the compiler to consult when processing files. This allows the fast import of modules.

You import the module via the `import mymodule;` declaration. You do need to put angle brackets around the name of your modules. In addition, you should use a semicolon at the end of the expression, which is not necessary when you use a C preprocessor.

Unlike header files, the name that you use to `import` a module is not based on the name of the file that contains the module interface.

It is based on the name that is declared in the `export module mymodule;` declaration.

Next, after the import module, you can only use these exported entities. Example:

```
import <iostream>;
import simple;

int main() {
 std::cout << "a squared: " << square(a) << "\n";
}
```

When you import a module into a file that is not part of the same module, you do not implicitly inherit all imports from the module interface file. In other words, when you import a module, you do not implicitly gain complete access to all other modules that that module relies on.

The C preprocessor behaves completely opposite.

If you want for some reason to provide users with a means to automatically import modules under which your module depends, you should add `export` in front of an import declaration in module interface files. You can do it in the following way:

```
export import <string>;
```

Then any file that imports this module implicitly inherits all import declarations that are exported from that module as well.

Splitting Modules

This is the last topic regarding the modules that we are going to cover.

Simulating Submodules. C++20 does not formally support the concept of nested submodules, but they can be simulated quite easily. The possibility to use a dot (`.`) in the names of modules allows adapting it to make it

easier to see the relation between modules and their submodules. Dots were explicitly allowed in module names to facilitate such hierarchical naming.

Module Partitions. Submodules can be imported individually by the rest of the application because they are just modules.

But *module partitions* are not.

They are only visible within a module.

```
// internals.cpp – Module implementation file for the internals partition

// 1. The module declaration of a module partition file is almost similar to that
of any other module file
// 2. Except that the name of the partition is added after the name of the module,
separated by a colon.
// 3. Except it is not allowed to export any entities

module mymodule:internals;

unsigned int from_c(char c)
{
//...
}
```

Then to use the internals module partition and its function in the module implementation file you have to import the partition.

```
// mymodule.cpp – Implementation
module mymodule;

// Module partitions can only be imported into other files of the same module, and
only via using import :partition_name;
import :internals;

unsigned int from_abc(std::string roman)
{
  // Use from_c(char) from the :internals partition.
}
```

Outside the module, a partitioned module can only be imported as a whole (through `import mymodule;`). Individual partitions can never be imported outside of the module.

Split Module Interface. You can partition the Module Interface as well:

```
//a.cppm
export module mymodule:parta;
//...
```

```
//b.cppm
export module mymodule:partb;
//...
```

```
// mymodule.cppm – Primary module interface file
export module mymodule;
export import :parta;
export import :partb;
```

Templates

A **template** describes a skeleton of code for a set of related classes or related functions.

When you instantiate a template with a given set of template arguments, the compiler generates a new definition of the class or template function based on the provided template arguments.

Creating a new definition of a function, class, or member function from specified template arguments for the template class or template function is called **template instantiation**.

> Templates are a unique feature of C++. B. Stroustrup said in the past, "Templates are not Generics (from C# or Java). Generics are primarily syntactic sugar for abstract classes. With generics (whether Java or C# generics), your program precisely defines interfaces and typically pays the cost of virtual function calls and/or dynamic casts to use arguments."

Template Parameters

Three types of template parameters can be used during template class or template function definition:

1. *Type parameters.* With function or class templates, you can introduce template type parameters either using the `class` or `typename` keyword.

2. *Non-type parameters.* In addition to type, template parameters include Non-Type parameters, which can have the type of:

- Integral constant
- Reference, and a pointer to a given function. Pointers to data of a function must have an external link type.

Starting from C++20, a template parameter can be of any fundamental type:

- bool
- float
- int
- enumeration type
- pointer type

- and reference type. [4, p.380].

However, in reality, at the moment of 2024, this is not supported by actual compiler implementations (please be careful). Also, the compiler needs to be able to evaluate the arguments corresponding to all non-type parameters at compile time.

3. *Template-template parameters.* A template can be customized with a template class. Details are available [1], Appendix B. This is a bit of an advanced construction to use another template provided as a "type" to use it inside our main template construction, but it's available. Example:

```
template<template<class, class> class H, class S>
struct T {
   H<S, S> my_pair;
};
T<std::pair, int> aa;
```

Template Syntax Remarks

Starting from C++11, it is not necessary during instantiating a template to write a space in the right shift operator `>>` when it's used during template instantiation. In the rare cases when you need to specify an integral type in a template parameter and wish to perform right-shift `>>`, you should enclose the expression in parentheses starting from C++11. Example of using `>>`:

```
template <class T, int size>
class AA {
};

AA<std::vector<int>, 2> obj;
```

The name of a *template class* or *template variable* or a *template function* augmented with the list of parameters for the template between angle brackets is called the **template-id** (See cpp reference documentation).

During the definition of a member function or a static member of a template class, there are two aspects:

1. First, it is not necessary to use the full **template-id** within a template definition.

   Example in which both `class ClassA` and `class ClassB` are valid, but `ClassA` is shorter:

   ```
   template <typename T>
   class ClassA
   {
   public:
       ClassA(size_t size) {}
       static void print(const ClassA& ctr) {}
   };
   ```

```
template <typename T>
class ClassB {
public:
    // Valid, but longer
    ClassB<T>(size_t size) {}
    static void print(const ClassB<T>& ctr) {}
};
```

2. Second, if you are defining methods of a template class outside the template class definition, then the name of the class must be qualified with **template id**.

   However, syntactically, after passing the scope resolution `::`, there is no reason to use a fully qualified **template id**.

   Example:

```
template<class Tx, class Ty> class MyPair {
private:
    Tx first;
    Ty second;
public:
    MyPair(const Tx& theFirst = Tx(),
           const Ty& theSecond = Ty());
    void f(){}
};

#if 1
template<class Tx, class Ty>
MyPair<Tx,Ty>::MyPair(const Tx& theFirst,
                      const Ty& theSecond)
: first(theFirst), second(theSecond) {}
#else
// Obsolete and too long.
// However, syntactically, it is correct as well.
template<class Tx, class Ty>
MyPair<Tx, Ty>::MyPair<Tx, Ty>(const Tx& theFirst,
                               const Ty& theSecond)
    : first(theFirst), second(theSecond) {}
#endif
```

Now, let's talk about full template function specialization. It's possible to use different syntaxes for function specialization from longer to shorter syntax (See A->B->C below).

On the complex rules of the admissible derivation of template arguments, functions see ([1], 13.4).

```
template <class T>
void swap(T* a, T* b)
{
```

```
    Ttemp = *a;
    *a = *b;
    *b=temp;
}
//A: full template-id
template<>
void swap<float>(float* a, float* b)
{
    float temp_temp = *a;
    *a = *b;
    *b = temp_temp;
}
//B: template id with missing all arguments in <>
template<>
void swap<>(double* a, double* b)
{
    double temp_temp = *a;
    *a = *b;
    *b = temp_temp;
}
//C: specialization which in fact even skip <> in template-id of swap function
template<>
void swap(int* a, int* b)
{
    int temp_temp = *a;
    *a = *b;
    *b = temp_temp;
}

int main() {
    int a = 0, b = 0;
    swap(&a, &b);
    return 0;
}
```

Finally, we would like to mention that the default template argument type must occur at the end of the class or variable template.

> However, there is one subtle thing about function templates. For function templates, strictly speaking, it's not a requirement that the default template argument type must occur at the end. This is so because a rich deduction template mechanism exists for function templates.

Template Instantiation

Templates can be instantiated in two forms:

Implicit instantiation.

The compiler will generate a specialization for the template only when it needs the definition. For example, the compiler instantiates a class template as a result of a definition of an object of a specific template instantiated type.

All member function definitions in the class template are templates. When the compiler needs the code for member functions of the class template, it generates it.

If the member of the template class is not needed by the program, then the compiler will not create instances of a member function or anything that is not required.

Example of implicit template class instantiation:

```
MyPair<int,int> a;
```

Explicit instantiation.

Explicitly instantiating a class template generates the complete class type definition and instantiates all member functions.

This happens regardless of whether you call the member functions or not.

You can explicitly instantiate:

- classes
- class methods
- template functions

Syntax looks like a class (class method, function) declaration, but with the extra `template` word. Example:

```
#include <iostream>

template <class Tx, class Ty>
struct MyPair
{
    Tx x;
    Ty y;

    void f(){}
};

template <class T>
void f(T&)
{}
template <class T>
void g(T&)
{}

template class MyPair<int, int>;     // explicit instantiation of all class
members
template void MyPair<int,int>::f();  // explicit instantiation of a class function
member
template void f<int>(int&);          // explicit instantiation of a function
template void g(double&);            // explicit instantiation of a function
```

```
int main() {
    return 0;
}
```

Essentially, the explicit instantiation of a class template instantiates all members. Explicit instantiation can be used to quickly test whether a class template and all its member templates can be instantiated without compile-time errors.

Variadic Templates

Variadic templates were introduced in C++11.

The motivation is to have the ability to work with a varying number of template parameters during template definition.

The smallest example of template function definition with variadic template arguments:

```
template <class...T>
void f(const T&...arg){}
```

Example of performing *printf* in variadic template style. The example is typical and can be found as an example using variadic template, e.g. here question in the StackOverflow:

```
#include <iostream>

// Function that takes zero arguments after expansion
void my_printf(const char* s)
{
    while (*s)
        std::cout << *s++;
}

template<typename T, typename ... Args>
void my_printf(const char* s, T value, Args ... args)
{
    while (*s)
    {
        if (*s == '%')
        {
            if (*(s + 1) != '%')
            {
                std::cout << value;
                s += 2;
                my_printf(s, args...);
                return;
            }
            ++s;
        }
        std::cout << *s++;
```

```
    }
}

template<typename ... Args>
void my_print_arg_pack_size(Args ... args)
{
    std::cout << "Number of elements in template arguments parameters pack: " <<
sizeof...(args) << '\n';
}

#if __cplusplus >= 201703L
    template<typename... Args>
    bool allLeftFold(Args... args) { return (... && args); }
#endif

int main()
{
    // FOR MSVC use compiler flag to define __cplusplus correcly: /Zc:__cplusplus

    std::cout << "C++ Standard: " << __cplusplus/100 << '\n' << '\n';

    // SINCE C++11
    my_printf("%i %i\n", 12, 12);
    my_print_arg_pack_size(1, 12.0, 'c');

#if __cplusplus >= 201703L
    // SINCE C++17
    std::cout << allLeftFold(1, 1, 0, 1);
#endif

    return 0;
}
```

As you see, there are a lot of constructions that use ellipses or dots `...`. There are two important things about variadic templates:

First thing: In the variadic template for ellipses token <`...`> you can put spaces anywhere around it. Construction <`...`> is used for various purposes in the context of variadic templates, which we will look at shortly.

Second thing: The only way to get the next template argument is to recursively call a template function whose template parameters are specified as `<TExtractType, RestTypes...>` via specifying some parameters in the usual way and some parameters via **unpack construction** (see below).

> If you have ever seen functional programming languages, this construction looks close to that.

Packing operations:

- `class ...Args` - **template parameters parameters pack** with optional name. Non-type template parameters can also be organized in the pack via a similar syntax `int...`. One more time - whitespace around <`...`> does not matter for the compiler. These are variadic template arguments for class

functions or methods. This construction means semantically there is a list of types `class Arg1, class Arg2, etc.` at this moment.
- `(Arg&&....params)` - **template arguments parameters pack** or **function parameters pack**. These are variadic arguments for template functions or methods. This construction means semantically there is a list of arguments `Arg1&& params1, Arg2&& params2, etc.` at this moment.

Unpacking operations:

- `args...` - **unpack** parameters (or **pack expansion**). Used inside the body of the template function or template class member function to semantically unroll all template-type parameters or arguments. Semantically, it means `arg1, arg2, etc..`.
- `sizeof...(args)` or `sizeof...(Args)` - the size of parameters in terms of the number of elements in Args. (**It is not the size in bytes**). See also documentation in cppreference.

Variadic templates have been introduced in C++11, but there is one more trick for variadic templates with the name **fold expression** introduced in C++17.

The syntax for the fold expression:

```
template<typename... Args>
bool allLeftFold(Args... args) { return (... && args); }
// Unary left fold (... op E) becomes (((E1 op E2) op ...) op EN)

template<typename... Args>
bool allRightFold(Args... args) { return (args&& ...); }
// Unary right fold (E op ...) becomes (E1 op (... op (EN-1 op EN)))
```

The allowable operation for folding includes all logical, arithmetic, bitwise, and comparison operators supported for 32-bit integer numbers.

So, essentially, a binary operator is used to use `args[1]` and replace `...` recursively into `( expression )`.

Documentation:

- cpp reference details about folding
- cpp reference details about parameter pack
- cpp reference details about "sizeof..." operator

Template Specialization

We have already quickly looked into template specialization. But now, we will look more closely into this concept.

The process of overriding the default behavior for template instantiation is called *specialization*.

An explicit specialization declaration must appear after the declaration of the primary/general template, at least due to the language requirements. Maybe some toolchain will allow you to violate this requirement, but it's not portable across C++ compilers.

In C++, you can specialize the following things:

1. Function template
2. Class template
3. Member function of a class template
4. Member class of a class template
5. Static data member of a class template
6. Template member function of a class template
7. Member class template of a class template

A template specialization can be:

- **Complete.** For example, specialization can be made for some concrete types. A complete class template specialization is a class definition. It starts with `template<>`, and it is not an empty class template. An empty `template<>` is syntactically reserved for explicit specialization and cannot be used to create a template without parameters.
- **Partial.** A partial specialization is a specialization of templates for specific cases. The partial specialization of the original template is a template. The template uses type `T` as the basis for something else. Partial specialization only works for classes but doesn't work for template functions. For template functions to obtain behavior similar to the partial specialization, you can combine template function definition with function overloading. A partial specialization has both (i) a template argument list and (ii) a template parameter list. The compiler uses partial specialization if its template argument list matches a subset of the template arguments of a template instantiation.

Example of complete function specialization (1):

```
#include <iostream>

template <class T>
void f(T arg) {
    std::cout << "general\n";
}

template <>
void f(int arg) {
    std::cout << "int\n";
}

template <>
void f<float>(float arg) {
    std::cout << "float\n";
}

int main() {
    f(1);
    f(1.0f);
    return 0;
}
```

Function templates can be fully specialized but not partially specialized, as we already have.

Example of partial and complete class specialization (2):

```
template <class T>
struct Single {
  Single() : var(T()) {}
  T var;
};

template <class T>
struct Single<T*> {
  Single() : var(T()) {}
  T var;
  int extra_for_ptr;
};

template <>
struct Single<int> {
  Single() : var(int()) {}
  int var;
  int extra_for_int;
};

int main() {
  Single<int> a;
  a.extra_for_int = 1;
  Single<int*> b;
  b.extra_for_ptr = 1;

  return 0;
}
```

Example of specialization for static data member of a class template (5)

```
template <class T> struct Single
{
    Single() : var(def_val) {}
    static T* CreateInstanceOfT();
    T var;
    static T def_val;
};

template <class T> T Single<T>::def_val(0);
template <>        int Single<int>::def_val(1);
template <class T> T* Single<T>::CreateInstanceOfT()  {return 0;}

int main(){
    Single<int>::CreateInstanceOfT();
    return 0;
}
```

Templates Miscellaneous

The general template must be declared before any partial or complete specialization. Each instantiated class template specialization has a copy of any static members. You may explicitly specialize static members.

In a function template specialization, a template argument is in fact optional, but if the compiler can deduce it from the type of the function arguments.

You can nest member templates within many enclosing class templates. If you explicitly specialize a template nested within several enclosing class templates, you must prefix the declaration with `template<>` for every enclosing class template you specialize. Example:

```
#include <iostream>

template <class T>
class Outer
{
public:
    template <class V>
    class Inner
    {
        public:
            static void f( T in )
            {
                std::cout << "general-template";
            }
    };
};

// Specialization of a member function of a class template nested in another class
template. Number (3) in the list of what is possible to specialize: Member
function of a class template.
template<>
template<>
void Outer<double>::Inner<int>::f( double in )
{
    std::cout << "specialisation-1";
}

// Specialization of a class template in another class template. Number (4) in the
list of what is possible to specialize: Member class of a class template.
template <>
class Outer<int>
{
public:
    template <class V>
    class Inner
    {
        public:
            static void f( int in )
            {
                std::cout << "specialisation-2";
```

```
            }
    };
};

int main()
{
    Outer<int>::Inner<int>::f(1);
    return 0;
}
```

Importantly: **You cannot explicitly specialize a class template unless its enclosing class templates are also explicitly specialized.**

An example of `template` for creating a specialization for the array with a fixed size.

```
#include "stdio.h"

template<class T>
void p(T a){ puts("not array"); }

template<class T>
void p(T a){ puts("not array"); }

template<class T, int sz>
void p( T (&t)[sz] ) { puts("array"); }

int main(int argc, char *argv[]) {
     intq[3];
     p(q);
     return 0;
}
```

Sometimes, while using templates inside templates, there is a need for the `template` qualifier in case of difficulty understanding from the compiler side what you want to call the *template member function*.

In this case, `template` plays the role of an extra marker for which you want to use the **template member function**.

In this sense, it's similar to `typename`, which is used in the context of having access to **type** members of template classes or structures.

```
template <typename S, typename T>
T* allocate(S& storage, int numberOfElements)
{
     T*res = 0;
     // res = storage.alloc<T>(numberOfElements);
     res = storage.template alloc<T>(numberOfElements);
     return res ? true : false;
}
```

> For more info, please view "B.13.6 template" in Special Edition, Bjarne Stroustrup.

Reference Collapsing Rules and Universal Reference

In C++, you can not declare a reference to a reference, but when reference collapsing occurs during template type substitution, the following rules are applied:

`T&&` => `T&` - valid from C++03 (1)

`T&, &&` => `T&` - valid from C++11 (2)

`T&&, &` => `T&` - valid from C++11 (3)

`T&&, &&` => `T&&` - valid from C++11 (4)

So, the C++ standard allows reference collapsing in templates.

Example for reference collapsing:

```
template <class T>
void f(T& val) {
   val = 0;
}
int i = 0;

//During compile-time
// f<int&>(int&&) => converts into => f<int&>(int&)
f<int&>(i);
```

Example of applying rule (4):

```
template <class T>
void f(T&& arg)
{}
```

If argument `arg` is LValue with type `Z`, the function will be deduced:

`f<T>(T&& arg) => f(Z& && arg) => f(Z& arg)` (see rule (3))

If argument `arg` is an RValue reference with type `Z&&`, the function will be deduced:

`f<T>(T&& arg) => f(Z&& && arg) => f(Z&& arg)` (see rule (4))

So, in fact, construction

```
template <class T>
void f(T&& arg)
```

```
{}
```

can bind everything.

The term used to describe that (by Skott Meyers) has obtained the name *universal reference*. Even though it's not official. We have already covered it, but now with the rules of reference collapsing, we hope it is now clearer what is going on.

Variants of Casting

const_cast - removes `const` and `volatile` modifiers from a pointer or reference.

static_cast - casting one type to another using standard or user-defined conversions explicitly.

dynamic_cast - the argument to dynamic_cast can be a null pointer - the result of this operation is a null pointer. The argument to dynamic_cast must be a polymorphic type. This requirement simplifies the implementation. The result of `dynamic_cast` need not be a polymorphic type. If it is impossible to understand what the type needs to be converted to (e.g., in the case of diamond-shaped inheritance with duplicates), a null pointer will be returned. It will also return a null pointer for the simpler base case when casting fails.

reinterpret_cast - force casting instructions for changing bit representation of integral (various forms of `int`) without reinterpreting bit patterns.

C-style type casting

- tries to use static_cast
- if not tries to use reinterpret_cast.
- if not, try to use in addition const_cast.

Private inheritance. In the context of casting, it's worthwhile to mention this as well. In addition to the invisibility of variables by the user of the derived class of members of the vase class, it also does not allow implicit typecasting.

```
struct A{int a;};
struct B:private A{int b;};

int main() {
  B b;
  { A& ba = b;      }  // error
  { A& ba = (A&)b; }  // ok
  return 0;
}
```

Concepts (from C++20)

Define Concepts

A concept is a named set of requirements.

It's possible, in general, to distinguish three categories of requirements [4,p.512] in the context of programming languages:

- Syntactic Requirements
- Semantic Requirements
- Complexity Requirements

The only requirements that we can express with C++ concepts are syntactic ones. Explicitly specifying any non-obvious semantic and complexity requirements of a concept right now should be done in the form of code comments or other documentation.

> In each area of CS, there are requirements beyond these three. But it's out of the scope of this note. These three categories are not a bad set.

The motivation for using the concept is to have the ability to provide you with a means to constrain the template type.

General things about concept:

1. Concepts are never instantiated by the compiler. They never give rise to executable code.
2. Concepts should be seen as functions that the compiler evaluates at compile time to determine whether a set of arguments satisfies the given constraints.
3. A *constant expression*, as usual, is any expression the compiler can evaluate at compile time.

Arguments of concept are specified via the sample `template <parameter list>` as for templatized functions and classes.

I. Atomic Requirement.

The first way to define concept is to express it via **Constraints**.

Example of concepts with a *"(1) Simple Constraint"*:

```
template <typename T>
concept Small = sizeof(T) <= sizeof(int);
```

In this example, the concept Small is defined with a single *constraint expression*. A single *Constraint Expression* is named atomic constraint.

The atomic constraints in the body of a concept definition can be combined with logical expressions connected with `&&` or `||` or `!`. Such constraint is called a *(2) Disjunction Constraint* and * (3) Conjunction Constraint*.

Next, you can define new *concepts* in terms of other concepts in a cascading style.

Example:

```
template <typename I>
concept Integer = SignedInteger<I> || UnsignedInteger<I>;
```

So, to define the concept in the form of a constraint, you should make it in the following way:

```
template <parameter list>
concept name = constraints;
```

Atomic, Conjunction, and Disjunction Constraint expressions are not the only way to define concept, but they're the first form of it and fundamental for more complex constraints.

To express the syntactical requirements, you should use requires expression. The **requires** specifies an expression on template parameters that evaluates a requirement (since C++20). Its syntax is the following:

```
requires { requirements }
```

or

```
requires (parameter list) { requirements }
```

All the compiler does with requirements is check whether they form valid C++ code. The concept defined via requires can have one of the following types:

II. A Simple Requirement.

Example of creating a concept using *requires* (type-4).

```
template <typename Iter>
concept RandomAccessIterator = BidirectionalIterator<Iter>
&& requires (Iter i, Iter j, int n)
{
  //int n; // Error: not an expression
  i + n; i - n; n + i; i += n;
  i < j; i > j; i <= j; i >= j;
};
```

A simple requirement consists of an arbitrary C++ expression statement, and **is satisfied** if the corresponding expression compiles without errors.

All variables you use in these expressions must be global variables or variables introduced in the parameter list of `requires`. You *cannot* declare local variables in the usual way inside `requires`.

III. A Compound Requirement.

Example:

```
template <typename T>
concept NoExceptDestructible =
requires (T& value)
{
  { value.~T() } noexcept;
  value +=1;
};
```

A *compound requirement* is similar to a simple requirement. But besides asserting that a given expression must be valid, a compound requirement can:

- Prohibit this expression from ever throwing an exception
- Constraint type that it evaluates to

Importantly (**and please be careful**), there is no semicolon after the expression inside the curly braces of a compound requirement. Using semicolons inside the compound requirement is a compile-time error.

The list of all possible types of compound requirements:

```
  // expr is a valid expression
  { expr };

  // expr is valid and never throws an exception
  { expr } noexcept;

  // expr is valid, and its type satisfies type-constraint
  { expr } -> type-constraint;

  // expr is valid and never throws an exception, and expr type satisfies the type
constraint
  { expr } noexcept -> type-constraint;
```

In this type of requirement, the body of a `requires` expression consists of a *sequence of requirements*, and each requirement is introduced with curly braces `{}`.

Each requirement ends with a semicolon, but one more time, all `expr` inside curly braces should not have semicolons.

IV. Type Requirement.

Example of type requirement constraints:

```
  template <typename S>
  concept String = requires
  {
    typename S::value_type;  // The requirement named value_type type is valid
type
  }
```

V. Nested Requirement.

Use a requirement inside the body of another requirement.

Example:

```
    template <typename S>
    concept String = requires
    {
      //Additional constraints in terms of  local parameters or input type
      requires Character<typename S::value_type>;
    }
```

See also cpp reference details.

Use Concepts

The concepts are used in template code.

The general outline for a constrained template with concept usage is as follows:

```
template <parameter list>
requires constraints
template body;
```

Use Concepts. Syntax A.

Example of using the concept to restrict template usage:

```
template <typename T>
requires MyConcept<T> && std::destructible<T>
const T& function(const T& a, const T& b)
{ return a > b ? a : b; }
```

In a template, the declaration keyword **requires** specifying the used constraint.

Use Concepts. Syntax B.

Also, there is another shorthand notation for using the concept:

```
template <typename S>
concept MyConcept = requires(S a, S b)
{
    a + b; // Summable
};

template <MyConcept T>
const T& function(const T & a, const T & b)
{
    return a + b;
}
```

Terminology about Concepts

- The **concept** -- is a named set of requirements.
- The **requirements** -- requires(){/..../};
- The **constraints** -- constraints are the types with which the template can be instantiated.

More Bigger Example Using Concepts

This example is inspired by Nicolai Josuttis's talk, "Back to Basics: Concepts in C++," presented at CppCon 2024 (https://www.youtube.com/watch?v=jzwqTi7n-rg).

It demonstrates:

- How to use concepts to articulate function overloading when the underlying container interface has fundamentally different APIs.
- How to apply both longhand and shorthand syntax for concepts.
- How to utilize concepts in conjunction with abbreviated functions templates.

```
#include <vector>
#include <set>

template <typename TCtr>
concept CtrHasPushBack =
requires(TCtr& a, typename TCtr::value_type value)
{
    a.push_back(value);
};

template <typename TCtr>
concept CtrHasInsert =
requires(TCtr& a, typename TCtr::value_type value)
{
    a.insert(value);
};
```

```cpp
//=========================================================================
// Lesson-1: Concepts can specialize template functions
template <typename TCtr, typename TItem>
requires CtrHasPushBack<TCtr>
void add_with_specialization(TCtr& ctr, const TItem& item) {
    ctr.push_back(item);
}

template <typename TCtr, typename TItem>
requires CtrHasInsert<TCtr>
void add_with_specialization(TCtr& ctr, const TItem& item) {
    ctr.insert(item);
}

//=========================================================================
// Lesson-2: Concepts should be seen as functions with bool
//           return value that the compiler evaluates at compile time.
static_assert(CtrHasPushBack<std::vector<int>>);
static_assert(CtrHasInsert<std::set<int>>);

//=========================================================================
// Lesson-3: Concepts using shorthand syntax
template <CtrHasPushBack TCtr, typename TItem>
void add_shorthand(TCtr& ctr, const TItem& item) {
    ctr.push_back(item);
}
template <CtrHasInsert TCtr, typename TItem>
void add_shorthand(TCtr& ctr, const TItem& item) {
    ctr.insert(item);
}

//=========================================================================
// Lesson-4: Fancy Style for adding concepts while using
//           Abbreviated Function Templates
void add_abbr(CtrHasPushBack auto& ctr, const auto& item) {
    ctr.push_back(item);
}
void add_abbr(CtrHasInsert auto& ctr, const auto& item) {
    ctr.insert(item);
}

int main()
{
    std::vector<int>a;
    std::set<int>b;

    add_with_specialization(a, int(2));
    add_with_specialization(b, int(2));

    add_shorthand(a, int(2));
    add_shorthand(b, int(2));

    add_abbr(a, int(2));
    add_abbr(b, int(2));
```

```
    return 0;
}
```

Predefined Concepts

The Standard Library defines a whole collection of predefined concepts defined in in the `std` namespace.

- *Core language concepts:* same_as, derived_from, convertible_to, integral, floating_point, copy_constructible, etc.
- *Comparison concepts:* equality_comparable, totally_ordered, etc.
- *Object concepts:* movable, copyable, etc.
- *Callable concepts:* invocable, etc.

Coroutines (C++20)

The execution of the usual function includes the ability to perform the following actions in the underlying platform:

- Create a stack frame
- Execute function
- Return value
- Delete stack frame

Coroutines are conceptually the same as functions, but they allow two extra actions:

- Suspend execution
- Resume execution

Coroutines are stackless. They reuse the stack frame of the caller to resume execution after a return. In another language, such as Python, the closest thing to C++ coroutines is *generator*.

In C++, the `return` statement is prohibited in a coroutine; instead, to finish the execution of a coroutine, you should use `co_return`, and it is the only correct way to return from a coroutine. Falling to the end of the coroutine without using `co_return` is equivalent to using (or not using) `return` for usual functions, which is only allowable for functions with a `void` return type.

Example:

```
task<int> func(int a, int b)
{
  co_return x;
}
```

`co_yeild` - returns a value from the coroutine to the caller. Suspend execution of the coroutine until subsequently calling the coroutine.

`co_return` - complete execution returning a value from coroutines.

`co_await` - suspend a coroutine while waiting for another computation to finish.

Containers

In the standard library, there are several built-in containers. Below, we will list them and provide references to the original documentation.

- std::array - statically sized array.
- std::vector - a dynamic array with dynamically allocated elements.
- std::list - stores values in a doubly-linked list.
- std::forward_list - singly-linked list of elements.
- std::deque - despite its name from theoretical Computer Science (double-ended queue), in fact, in C++, it is a hybrid data structure. Typical implementations use a sequence of individually allocated fixed-size arrays.
- std::set - a container in which each element can appear only once. To insert and remove elements use insert() and erase(). The std::set can store any elements that can be ordered. By default, the ordering of all elements is done with `<` operator. If you need to have the ability to store several items in the set, you should use std::multiset. Sometimes such container is named *multiset* or *bag*, because in fact, they violate the mathematical definition of the *set*. This container may contain the same element more than once.
- std::unordered_set - a container in which each element can appear only once, similar to std::set. API is also similar. However, std::unordered_set does not utilize order relation and instead uses hash functions to provide set API. If you need functionality that will allow you to store several of the same items that do not have an order relation, you may be interested in std::unordered_multiset
- std::map and std::unordered_map - Associative array. Keys in a map need to be unique. And each key in containers is associated with a single value.
- std::multimap and std::unordered_multimap- map containers which allows multiple values to be associated with the key.

Acknowledgements

Konstantin Burlachenko, would like to acknowledge:

- Editors of that document Vadim Sofin, Mikhail Filimonov for producing useful feedback. Editor Dmytro Ovdiienko for producing very big feedback, constructive suggestions, and his big contributions to document improvement.

- Acronis company. In 2011-2012, it invited Dr. B.Stroustroup to give a long one-day talk for Acronis, HQ, and Parallels, HQ in Moscow employees. Dr. B.Stroustroup gave an excellent and practical talk by appealed to everybody, which I highly appreciated.

- Yandex company that, around 2014, invited Dr. Scott Meyers to Yandex HQ in Moscow. He gave a pretty sophisticated three-day overview of several features of C++11 and some parts of C++14 for Yandex employees.
- Thanks to NVIDIA and HUAWEI, the places where I have used those languages a lot while creating complex software systems.
- Thanks to all my colleagues and friends from the past with whom I went through various steps of figuring out the subtle details of C and C++. Between 2010 and 2020, I created personal engineering drafts, some of which are available here (old home page K.Burlachenko). All C and C++ information from it has been evolved into that document.

Thanks to Dr. B.Stroustroup. He created a language that is used to create the most important software in the world. Interestingly, this language has an artificial force that makes people better in all aspects of science and engineering.

The quote that I have learned from Tim Roughgarden is the following:

> Perhaps the most essential principle for the good algorithm designer is to refuse to be content - Aho, Hopcroft, Ulman, 1974

The mindset that C++ is shaping helps to look into details and abstract when needed. That kind of mindset can not be obtained for free, so it's part of why C++ is complicated. I believe such a mindset is one of many reasons that helped me join the Laboratory of Prof. Peter Richtarik and helping me in producing research. The lab conducts world-level research in math optimization and Machine Learning. The Lab is a part of the KAUST AI Initiative led by Jürgen Schmidhuber.

References

[1] The C++ Programming Language: Special Edition, B.Stroustrup

[2] C: A Reference Manual, 5th Edition. Samuel Harbison, Guy Steele Jr.

[3] The continuing evolution of C++. Bjarne Stroustrup. Presentation

[4] Beginning C++20. From Novice to Professional. Sixth Edition. Ivor Horton, Peter Van Weert

[5] C++17 Standard Library Quick Reference. Peter Van Weert and Marc Gregoire

[6] ISO/IEC 14882:2003 Programming languages — C++

[7] C++ language Reference. cppreference.

[8] Bjarne Stroustrup's C++ Style and Technique FAQ

[9] Bjarne Stroustrup's C++11 FAQ

[10] Useful catalog of resources from CppReference: Standards, ABI

[11] Compiler Explorer. An interactive online compiler which shows the assembly output of compiled C++, Rust, Go

[12]CS 5220 Applications of Parallel Computers by David Bindel.

[13] Bit Twiddling Hacks by Sean Eron Anderson (seander@cs.stanford.edu)

[14] Hacker's Delight. Second Editiond by Henry S. Warren, Jr.

[15] MIT.6.172 Performance Engineering Of Software Systems, 2018. Prof. Charles Leiserson, Prof. Julian Shun

[16] Instruction tables By Agner Fog. Technical University of Denmark, 1996 – 2022.

[17] Windows via C/C++ (softcover) Fifth Edition by J.Richter, C.Nasarre, 2011

[18] Beginning C++23: From Beginner to Pro

Appendices

Virtual Inheritance Inside

The detailed document with some details regarding this topic in the GCC compiler was previously available on the website http://phpcompiler.org/articles/virtualinheritance.html. However, right now this website is not available.

Regarding the implementation of virtual inheritance inside the MSVC compiler, please take a look at this archive article http://www.openrce.org/articles/files/jangrayhood.pdf with the title "C++: Under the Hood" by Jan Gray, March 1994. Jan Gray is a Software Design Engineer in Microsoft Visual C++ Business Unit.

The first question that we're going to address is what program #1 will output:

```
#include <stdio.h>
class A{};
class B1: virtual public A{};
class B2: virtual public A{};
class B3: virtual public A{};
class B4: virtual public A{};
class FROM_B1_B2 : public B1, public B2{};
class FROM_B1_B2_B3_B4 : public B1, public B2, public B3, public B4{};

#define PRINTSIZE(class_name) printf("sizeof class '" #class_name "' is equal to
%u bytes\ n", sizeof(class_name))

int main(int argc, const char** argv)
{
  PRINTSIZE(FROM_B1_B2);
  PRINTSIZE(FROM_B1_B2_B3_B4);
  return 0;
}
```

In this program, we measure the size of the class by using the `sizeof` operator. This information about the sizeof operator can be e.g., obtained from ANSI ISO IEC 14882 C++ 2003, 5.3.3:

> The sizeof operator yields the number of bytes in the object representation of its operand. The operand is either an expression, which is not evaluated, or a parenthesized type-id... When applied to a class, the result is the number of bytes in an object of that class, including any padding required for placing objects of that type in an array...

The hidden fields of the objects and padding are stored and taken into account while applying `sizeof`. So, an honest answer is that the size of the object is implementation specific, but in most systems, objects of class `FROM_B1_B2_B3_B4` will occupy $4 \cdot sizeof(pointer)$ bytes, and objects of class `FROM_B1_B2` will occupy $2 \cdot sizeof(pointer)$ bytes.

The essence of type casting in C++ is shifting the memory window so that the data of the base type that we look at in the memory layout would be identical to what we would look at if we created an object of the base class, without a derived type. Such a thing can be accomplished if the sequence of class inheritance is linear and the memory for the object of each class is just stacked to the tail of the base class object. You can play with the following code snippet and observe that more likely your toolchain will generate the same address:

```
#include <stdio.h>

#define PRINTADRESS(lvalue) printf("adress of '" #lvalue " is: %08X\n", (void*)&
(lvalue))

class Simple
{
public:
  int base1;
  int base2;
};

class SimpleBased : public Simple
{
public:
  int der1;
  int der2;
};

int main(int argc, const char** argv)
{
  SimpleBased simple_based;
  PRINTADRESS(simple_based.der1);
  PRINTADRESS(simple_based.der2);
  PRINTADRESS(simple_based.base2);
  printf("NOW IT IS MORE LIKELY THAT ALL ADDRESSES WILL BE THE SAME\n");
  PRINTADRESS(simple_based.base1);
  PRINTADRESS(simple_based);
  PRINTADRESS((Simple&)(simple_based));
  return 0;
}
```

But with using virtual inheritance, for example, diamond-shaped, common data is created for two inheritance paths from the base class.

```
#include <stdio.h>

#define VIRTUAL virtual

#ifndef VIRTUAL
    #define VIRTUAL
#endif

class Top
{
public:
    // ~Top() { printf("~Top()\n"); }
    Top() { printf("Top()\n"); }
    Top(int) { printf("Top(int)\n"); }
};

class Left: VIRTUAL public Top
{
public:
    // ~Left() { printf("~Left()\n"); }
    Left()
    {
        printf("Left()\n");
    }

    Left(int a) : Top(a)
    {
        printf("Left(int)\n");
    }
};

class Right: VIRTUAL public Top
{
public:
    // ~Right() { printf("~Right()\n"); }
    Right()
    {
        printf("Right()\n");
    }

    Right(int a) : Top(a)
    {
        printf("Right(int)\n");
    }
};

class Bottom: public Left, public Right
{
public:
    // ~Bottom() { printf("~Bottom()\n"); }
    Bottom()
```

```
    {
        printf("Bottom()\n");
    }

    Bottom(int a) : Left(a), Right(a)
    {
        printf("Bottom(int)\n");
    }
};

int main(int argc, const char** argv)
{
    Bottom b(123);
    return 0;
}
```

In the example above, we use the `virtual` keyword that should be used for inheritance branches that come from a common ancestor for different inheritance paths. An example represents classical diamond shape inheritance. In such a situation with such a memory layout, it is less possible to create a simple strategy with a sliding window that looks "*beautifully*" at the given object and during upcasting and downcasting shifts the windows. To mitigate this, the following trick is used.

In our original example with several `B` classes in fact for each `B1,B2,B3,B4` class the compiler to provide access to the shared data in `class A` creates a field of type `vptr.B1, vptr.B2, vptr.B3, vptr.B4`. All these pointers are pointed to the `vtable` table.

For non-virtual function calls, the member function to call is statically determined at compile time. For virtual functions, a derived class can override, or replace, the actual function definition to be used when an inherited member.

Whether the `override` should be used for a specific class depends upon whether the member function is declared `virtual`. Firstly, for each class, we construct a Table of virtual functions. A `vptr` pointer is added to a class instance to address the virtual function table (`vtable`).

Each virtual function in a class has a corresponding entry in that class's vtable. Each entry holds the address of the virtual function override appropriate to **that class**. Calling a virtual function requires:

- Fetching the instances `vptr`.
- Indirectly calling.

Each virtual table consists of:

1. **virtual base offset** - number of bytes to be added to the `vptr.B1` field address to get the address of shared variables from `class A`.
2. **top** - offset of the `vptr.B1` field from the beginning of the structure. This may be required for complex situations with casting.
3. **std::type_info** - value which stores information about the actual type which can be queried in runtime via typeid.
4. If there are pointers to real called functions tables `sizeof` class 'FROM_B1_B2_B3_B4' is equal to 32 bytes, if the size of a pointer is 8 bytes.

5. **Table of virtual functions** - is essentially an array of pointers to (virtual) functions. These pointers are used at runtime to invoke the appropriate function. At compile time, it may not yet be known which function should be called. A derived one is implemented in a class that inherits from the base class or a base class.

The C++ standards do not specify exactly how dynamic dispatch is implemented. In addition, the compiler creates a separate virtual method table for each class. When an object is created, a pointer to this table (vptr) is added as a hidden member of the class.

The compiler must also generate "hidden" code in the constructors of each class to initialize a new object's virtual table.

A method access specifier determines whether you can call it or not outside the class. The access modifier does not play a role in whether you can override it or not.

Object Orientated Design

C++ is a multi-paradigm language. A big part of it is devoted to object-oriented programming (OOP). OOP can be implemented in different ways. In this section, we will share background about the subject. These materials are based on vision T.Hoare from 1965-1970.

In the context of OOP:

- *Type* - is a set of objects with operations on them.
- *Polymorphism* - literally translated as "many forms". In the context of OOP, it means working with objects of different shapes in the same way.

From which place does OOP come?

OOP was born from the shortcomings of structured programming. Structure programming essentially means C-style development when there are structures and functions. Advantages of structured programming:

- Errors are coming at the early stages of development.
- Distribution of work.

Disadvantages of structured programming:

- Any modification changes the structure of data types.
- Any modification changes the structure of the logic of the function. The underlying reason is that data is located in the lowest level of hierarchical calls.

To the best of my knowledge, one of the authors (Konstantin Burlachenko) basic concepts of OOP were strongly born in 1966 by Tony Hoare, even though the first thinking about this may be from 1950-1960. In 1966, Tony Hoare gave lectures on sharing records and proposed:

1. Select actions on data and use an add-on to modify data. This has been called **encapsulation.**

2. To modify software entities, if possible, use an add-on over existing entities. It's been called **inheritance.**

3. T.Hoare proposed to make the interaction in the program of objects the same as in the real world and singled out:

    - **accessory interaction**, which is like reading the characteristics of an object. This type of interaction is sometimes called **static** in programming languages.
    - **event interaction**, which is an interaction associated with a change in the state of an object. This type of interaction is called dynamic.

Next, there are several other concepts that have been formalized during the birth of OOP:

- An *object* is a concrete implementation of an abstract type that has the characteristics of state, behavior, and identity.
- The *behavior* is a description of an object in terms of changing its state and the transmission of messages in the process of exposure to or under the influence of other objects.
- The *individuality* is the essence of an object that distinguishes it from other objects

Relationships between objects:

- *Relationship of use* or *Delegation*. An object can play the role of influence, performance, and mediation.
- *Inclusion.* An object can be composed of other objects:
    - *Aggregation by reference* - a situation when an object references parts.
    - *Composition* - a situation when an object consists of parts.

Categories of objects in OOP:

- *Real objects.* An abstraction of the actual existence of objects in the physical world (table).
- *Roles.* An abstraction of purpose, purpose (student, person).
- *Incidents.* An abstraction of something that happened (power surge)
- *Interaction.* Objects obtained from the interaction between objects
- *Specifications.* Used as a rule representation or quality criterion

What is Class:

A class is an abstraction of things in the real world, such that all items in this set have the same characteristics. All objects are subject to and conform to the same set of rules and behaviors.

Relationships between classes

1. *Inheritance.* One class is built on top of another
2. *Usage.* An object of one class calls methods on an object of another class
3. *Include.* A class includes other classes.
4. *Meta-Classes.* A class whose objects are classes.

What is Domain:

A domain is a separate real hypothetical world inhabited by a distinct set of objects that behave according to the rules or behaviors characteristic of the domain. The class is defined in one domain. A class in a domain can only require the existence of other classes in that domain.

Now you know what *Object object-oriented design* is. To the best of our knowledge, it's more an art than a science.

Object-Orientated Design Patterns

In the book Design Patterns (Gamma, Helm, Johnson, Vlissides) 2011" in the preface, authors say: "*You shouldn't have to go to the dictionary to explain the terms type and polymorphism*".

In our technical note, we have covered this in the previous section Object Orientated Design without limiting the amount of readers.

Whether *Design Patterns* are needed or not, it's not bad to be aware of them. Unfortunately, when Software Projects contain a lot of them, it is impossible to work with a project to make actual work and create actual algorithms.

Main Theses:

1. Program according to interfaces, not according to implementation
2. Prefer *composition*(another term is *inclusion*) over class inheritance. With delegation (another term is *use*), the composition becomes as powerful a concept as an inheritance.
3. The *type parametrization* after inheritance and composition is one more way to solve a problem in the OOP style.
4. The patterns can be useful when reorganizing reusable software.

List of patterns that are described in the book:

Pattern	Meaning
Abstract Factory	Create a family of related objects.
Factory	Defines an interface for creating a collection of objects. Interface methods are essentially Factory Methods in the form of *create**.
Builder	Construction of a complex object. The manager (director) is configured by the desired builder (builder). The director then tells the builder to build the part. The builder builds the part and adds it to the product. The customer picks up the product from the builder.
Factory Method (virtual constructor)	Create an instance of an object by its ID.
Prototype	You want to create an instance of the class, and you use the factory via, for example, `CreateInstance`. However, you cannot call `CreateInstance` from the class because it is unknown. One way to resolve this is for Objects to inherit the `Prototype` interface, which has a `Clone` method to clone itself, returning a Prototype, and from the obtained `Prototype` you can call `Initialize`.
Singleton	Ensures that the class has one instance.
Adapter(Wrapper)	Converts an interface from one class to another. Implementations: **(A1)** Openly inherited from the `Target` interface and closed from `Adaptee`. Thus, access to Adaptee will be only inside the adapter. Exploit this to implement `Target`. **(A2)** Openly inherited from the `Target` interface. Take a pointer to `Adaptee` in the constructor. When implementing the `Target` interface, exploit this object.

Pattern	Meaning
Bridge	Separate the implementation from the interface so that both can be developed in parallel. The implementation is similar to A2 for the `adapter`. It differs only philosophically. Also, the pattern interface can be more complete and provide higher-level work primitives (draw a square). While the implementation provides more low-level primitives (draw a line)
Composite	Tree structure of objects with parent-child relationships.
Decorator	Adds new responsibilities. The implementation is similar to `A2` of the Adapter. before/after drawing an object in the decorator, we draw a decorated rig and then call at the draw object being decorated.
Facade	Unified interface to the subsystem. Simple and intuitive interface for working with the system.
FlyWeight	Separation is used to support many small objects effectively.
Proxy	Proxy access to an object. For example, using this approach, you can obtain: (i) the creation of a heavy part of an object only on demand can be implemented; (ii) caching some calculations.
Chain of Responsibility	Links objects - recipients of a request into a chain, and passes the request along the chain in turn until it is processed. The chain can be organized by having a pointer to the next object, the recipient object in a chain list. Using the pattern, you can implement Help in the system.
Command	Wrap the command in an object. You can queue requests in this way, log them, and support the cancellation of operations.
Iterator	Just iterator similar to iterator in C++ Standart Library.
Mediator	System objects do not refer to each other but work through an intermediary entity. The pattern allows you to reduce the connectivity of objects in the system.
Memento	Fixes the internal representations of an object so that the state can be restored later.
Observer (publish-subscribe)	When the state of an object changes, all dependents on it are notified.
State	Variation of object behavior depending on the internal state. Models behavior based on state. The table method for a state machine focuses on defining transitions between states.
Strategy (Policy)	Allows the implementation of interchangeable algorithms. **Static polymorphism** can be implemented via a template parameter. A strategy instance is simply a member of an algorithm object. This member type is obtained from the template parameter. **Dynamic polymorphism** is implemented through virtual methods.

Pattern	Meaning
Template Method	There is an algorithm that consists of steps of **primitive operations**. The idea is to let subclasses override some of these **primitive operations**. The algorithm implemented in **template method** defines the skeleton of an Algorithm, but it relies on this primitive operation. In the context of this approach, we can recall the principle of Hollywood - "*Do not call us, we will call you ourselves*".
Visitor	An operation to be performed on each object in the collection. As a rule, the visitor has typed visit methods. When the container is traversed, the element of the container called has an interface `Accept` and the visitor passed into the element accepts as an argument. Then this `Accept` method calls the desired `visitSmth` method from the visitor, passing a reference to *this.
Allocator	Proxy to real storage

Several other design principles:

1. "*Functions placed at the lower levels may be redundant or of little value when compared to providing them at the higher level.*" Saltzer, 1984
2. The needs of the many (all the people reusing your class) outweigh the needs of the few (those who maintain your class's implementation) or the one (the class's original author). (Also known as Fun. Principle).
3. Brittle Base Class problem. What should a base class have to serve all conceivable derived classes without having any dummy data and methods?

Performance Optimization for general-purpose CPU

Based on materials from MIT 6-172, 2018.

Preparation

First, get the correct working code. Then, add preserving correctness regression testing. Interestingly, this lesson has been obtained in two different ways:

- From courses
- From experience working on big codebases

The compiler automates many low-level optimizations. To tell if the compiler is performing a particular optimization, look at the assembly code.

Performance Optimization

Here, we present materials presented by Prof. Charles Leiserson, Prof. Julian Shun, and invited speakers during the course [15] Performance Engineering Of Software Systems, 2018.

1. **Select a Good Algorithm.** The good asymptotically rate algorithm is not necessarily the fastest, however, it's a good start heuristic. In some cases, reducing the work does not automatically reduce the

algorithm's run time due to the complex nature of computer hardware, including instruction-level parallelism, caching, vectorization, branch prediction, etc.

2. **Pack structure into bits.** If you have a structure that stores integers with values, you can facilitate bitfields in C and in C++. This method saves space, but to extract and set bits, the read and write code will be coupled with bit operations.

   > Sometimes unpacking and decoding are the optimization, depending on whether more work is involved moving the data or operating on it.

3. **Data structure augmentation.** Motivation is to add information to a data structure to make common operations do less work.

4. **Precomputation.** The idea of precomputation is to perform calculations in advance to avoid doing them at "mission-critical" times. We precompute the table of coefficients when initializing and perform table look-up at runtime.

5. **Compile-Time Initialization.** The idea of compile-time initialization is to store the values of constants during compilation, saving work at execution time. One way is to create a code snippet that generates a function for you. (Such a technique is called *metaprogramming*).

6. **Caching.** The idea of caching is to store results that have been accessed recently so that the program does not need to compute them again.

7. **Exploit Sparsity.** The idea of exploiting sparsity is to avoid storing and computing on zeroes. Structures such as Compressed Sparse Row(CSR) can be used to compress information about sparse matrices, sparse graphs, and sparse weighted graphs.

8. **Constant folding and Propagation.** The idea of constant folding and propagation is to evaluate constant expressions and substitute the result in further expressions during compilation.

9. **Common-Subexpression Elimination.** Avoid computing the same expression multiple times by evaluating the expression once and storing the result for later use.

10. **Algebraic Identities.** Replace expensive algebraic expressions with algebraic equivalents that require less work.

11. **Short-Circuiting.** When performing a series of tests, stop evaluating as soon as you know the answer. The `&&`, `||` are short-circuiting logical operators from C and C++.

12. **Ordering Tests.** Perform those tests that are more often successful, or select an alternative test before tests that are rarely successful. Inexpensive tests should precede expensive ones (*Compilers can not do it right now*).

13. **Creating a Fast Path.** In Computer Graphics, people apply checks based on axis-aligned bounding Boxes firstly for simple geometric tests. Try to construct the same thing for your situation (*Compilers can not do it right now*).

14. **Combining Tests.** Replace a sequence of tests with one test or switch.

15. **Hoisting.** Hoisting (also called loop-invariant code motion) is to avoid recomputing loop-invariant code in the body of a loop.

16. **Sentinels.** Sentinels are special dummy values placed in a data structure to simplify the logic of boundary conditions and, in particular, the handling of loop exit tests (*Compilers can not do it right now*).

17. **Loop Unrolling.** Attempt to save work by combining several consecutive iterations of a loop into a single iteration. Another benefit of loop unrolling is that it opens more compiler optimizations to apply.

    - *Full loop unrolling:* All iterations are unrolled
    - *Partial loop unrolling:* Several, but not all, of the iterations are unrolled.

18. **Loop Fusion.** Loop fusion(also called jamming) is to combine multiple loops over the same index, saving the overhead of loop control. In addition, this technique is good for cache locality, and during loop fusion common subexpression may be obtained (*Compilers can not do it right now in full details*).

19. **Eliminating Wasted Iterations.** Modify loop bounds to avoid executing loop iterations over empty loop bodies.

20. **Tail-Recursion Elimination.** For recursive functions, replace a recursive call that occurs as the last step of a function with a branch, saving function-call overhead.

21. **Coarsening Recursion.** The idea of coarsening recursion is to increase the size of the base case and handle it with more efficient code that avoids function-call overhead (*Compilers cannot do it right now in full details*).

22. **Bit Tricks.** The `min` and `max` functions executed during conditions `if/else` can sometimes be expressed in bits and logical operations. If you are eliminating the unnecessary `if/else` statement, it is useful for low-level details of how the CPU works (namely, pipelining). Sometimes, the compiler does not perform such optimizations, and you must do it by hand. Materials about bit tricks are presented in [13], [14].

23. **Stack Frame Pointer Omission.** A *stack frame* is the area on the stack used by the current function, and it includes stack space for function arguments (sometimes called linkage block), return address, base pointer, local variables for the function code itself, and local variables for functions that are used in the current function. The frame pointer is implemented as a used register that is used to address local variables and arguments. If functions don't need a frame pointer for their operation and can use the Stack Pointer during their logic, the usage of the frame pointer can be omitted. It is the case if:

- The difference between the stack pointer and the frame pointer is a compiler-time constant
- It's constant in the function body in all code paths through the function
- Access to a local variable can be attained through the stack pointer

The benefit is this in this case - the compiler will have one more register in case of this optimization. To specify this in MSVC use /Oy for GCC/CLANG use -fomit-frame-pointer.

24. **Minimize Write from different Thread to the Same Memory.** Technically, if you have CPU threads that are assigned to different CPUs, you **can** write to the same memory location. However, from a memory perspective, when different logic is executed in different CPU cores, there are two hardware protocols that come into play in this situation:

- *cache consistency* protocol guarantees that all copies of DRAM cache lines are the same in all Caches in the system

- *cache coherence* protocol guarantees any read of memory returns the most recent update anywhere in the system

Technically, you can write in the same place (atomically). In reality, however, in real hardware, it leads to phenomena called an **invalidation storm**. It can lead to big performance bottlenecks in protocols such as MSI.

25. **Read after Read is not a race.** If two instructions perform a read and another performs a read as well from another thread, then in this case, there is no race condition. So, you can safely perform read after read without any synchronization. Another pair's read/write, write/read, write/write leads to race and nondeterministic behavior.

26. **If you are using parallel algorithms, then use work-efficient algorithms.** If you want to replace a sequential algorithm with a parallel version, then firstly, practical parallel algorithms should be work-efficient.

This property means that if the launch algorithm is in a single core, it should have the same asymptotic behavior in terms of computational complexity as a best (or good) sequential algorithm.

27. **Use Modern Compilers.** Compiler developers create new optimizations with each release.

28. **Eliminate as possible integer modulus and division.** The modulus division and integer division are typically expensive. The information about instruction latency (the delay in a dependency chain under the assumption that input data is available in the register file), throughput (the maximum number of instructions of the same kind that can be executed per clock cycle), required execution port for INTEL and AMD CPUs are available in this technical document [16].

29. **The order of linked files in some compilers can have a bigger effect than O2 or O3**. The order of linked files affects the placement of the generated code in the final binary file. Performance varies due to changes in cache alignment and page alignment. LLVM/CLANG has several compile options to handle code alignment: `-align-all-functions`, `-align-all-blocks`, `-align-all-nofallthru-blocks` (See this blog post).

30. **Try to precompute tables of values in compile-time or during runtime.** These methods have both their pros. and cons. If you compute values at compile time, you will need to put this information finally into the `data` segment of the binary image. Increasing the size of the `data` segment will increase the loading time.

    On the other hand, if the store has not yet computed values globally, because all uninitialized or initialized global variables by zero are stored in the `bss` segment - they do not require to be stored in the binary image. In this case, the binary image file loading time is decreasing, but you need to fill in values yourself at runtime.

31. **If you are using spinlock, be aware of yielding time for OS.** The cost of creating a thread in Linux/Posix is more than $10,000$ cycles. In Windows, the actual time for thread execution is about 30 milliseconds (roughly speaking $120,000,000$ cycles), and a system API call costs about 1000 cycles.

So, in Windows and Linux, if performance is the goal and you know that you will not block for a long time, maybe at least some amount of time, you should wait in a spinlock (at least 3000 - 4000 clocks) and after this yield time for the OS during spinning.

Sometimes, this kind of synchronization is denoted as *Competitive Mutex*. If the mutex is released during spinning, compute spinning is optimal, if the mutex is released after yielding, this algorithm 2 is optimal. There exists a randomized algorithm that can achieve a 1.58 competitive ratio.

32. **Force uses sequential consistency with memory fences.** To restore sequential consistency in the multithread program, you have to use a memory fence. Memory fences can be issued explicitly or implicitly by locking, exchanging, and some other synchronization commands. **The typical cost of a memory fence is comparable to L2-cache miss.**

33. **Consider the latency of memory accesses from different levels.** Take into account a multicore cache hierarchy and different times for memory access. The relative latency and throughput for access for CPU Caches and Disk space is the following:

- L1: 2 nanoseconds
- L2: 4 nanoseconds
- LLC/L3: 6 nanoseconds
- DRAM Memory: 50 nanoseconds. Throughput: 10 GB/s
- SSD: 10 000 - 20 000 nanoseconds. Throughput: 2 GB/s seq. access, 200 MB/sec random access.
- HDD: 10 000 - 20 000 nanoseconds. Throughput: 200 MB/s seq. access, 1 MB/sec random access.

34. **Use Cache-Efficient and Cache-Oblivious Algorithms.** One way to solve issues with caching is to use or create cache-efficient or cache-aware algorithms (Example: k-way merge sort).

    The harder alternative is to use or create cache-oblivious algorithms:

    - They do not have parameters to tune.
    - They do not have explicit knowledge of the cache size in the machine.
    - They handle multilevel caches automatically and are passively autotuned.

    Example of Optimal Cache-Oblivious Sorting via Funnelsort. A list of some Cache-Oblivious algorithms is presented in MIT 6.172 course lecture 15, 2018.

35. **Do not underestimate the algorithmic level.** Several tricks that can be done at the algorithm level:

    - Store pre-computed results
    - Early cutoffs
    - Use a clever pruning strategy if the algorithm has a combinatorial search nature

36. **If the algorithm has internal/meta parameters to tune, decide how to choose them.** Several strategies to decide internal parameters in the algorithm to use:

    - Construct the model of the computation device. It can explain exact magic constants in the algorithm, but it is hard to build, and it cannot model everything.
    - Heuristic-based solution. Works most of the time; however, it always has suboptimal performance.
    - Exhaustive search. The method tries all possible internal parameters of the algorithm and will find the optimal. It can take too long.

- Autotuning-based solution. Try parameters, and if satisfied, then finish; if not, select a new candidate heuristically.

37. **Be aware of the underlying reasons for slow speed.** The popular reasons for slow speed are the following:
    - *Insufficient parallelism.* You need parallelism to keep multicore, vector units, and GPUs busy.
    - *Scheduling overhead.* Too much parallelism is, at best, useless, but in fact, it even hurts performance due to scheduling overheads.
    - *Lack of memory bandwidth.* Faster memory reuse means better cache locality. Locality takes place at different levels L1, L2, and L3. To check if this is a reason for slow execution, run P identical copies of the serial code in parallel.
    - *Contention.* locking primitives for synchronization, true sharing, false sharing.

C++23 - How to Invoke C++2023 Compiler

Once the C++ standard committee adopted the first changes to C++23, the working name for this new version was named `C++2b`, which means "*probably to be C++23*".

Therefore, at least before the official (full) support of C++23 compiler-writers, use this flag to specify language standards.

- **GCC:** Gnu Compiler Collection. C++23 features have started to be available since GCC 11 (https://en.cppreference.com/w/cpp/compiler_support/23). However, better support of C++23 features has been available since GCC 13. The C++ status in GCC: https://gcc.gnu.org/projects/cxx-status.html

  ```
  g++-13 -x c++ --std=c++2b <filename>.cpp
  ```

  Some versions of GCC (such as 13.1) also support the C++23 standard flag:

  ```
  g++-13 -x c++ --std=c++23 <filename>.cpp
  ```

  Support for modules in GCC is enabled with the -fmodules-ts option.

- **CLANG:** The CLang is a language front-end and tooling infrastructure for languages in the C language family leveraging the LLVM back-end. The C++ status in CLang: https://clang.llvm.org/cxx_status.html

  ```
  clang++-17 -x c++ --std=c++2b <filename>.cpp
  ```

  ```
  clang++-17 -x c++ --std=c++23 <filename>.cpp
  ```

- **MSVC:** Microsoft Visual C++ (MSVC) is a compiler for C++ applications developed by Microsoft Corporation. At the moment of writing this technical note, there is no option `/std:23` from /std for MSVC. The C++ status in MSVC: https://learn.microsoft.com/en-us/cpp/overview/visual-cpp-language-conformance

```
:: Even to build simple code snippets, you should have a collection of
programs (toolchain) that will be used together to build various
applications for OS
::  https://learn.microsoft.com/en-us/cpp/build/building-on-the-command-
line?view=msvc-170

:: For compiler flags references:
::   https://learn.microsoft.com/en-us/cpp/build/reference/compiler-options-
listed-alphabetically?view=msvc-170
::   https://learn.microsoft.com/en-us/cpp/build/reference/compiler-options-
listed-by-category?view=msvc-170

:: Execute vcvarsall.bat from Windows SDK or Visual Studio.
:: Default Path for Visual Studio 2022

::call "C:\Program Files\Microsoft Visual
Studio\2022\Community\VC\Auxiliary\Build\vcvarsall.bat" x86_amd64
call "%VS2022INSTALLDIR%\VC\Auxiliary\Build\vcvarsall.bat" x86_amd64

echo **************IMPORTING MSVC TOOLCHAIN IS
FINISHED**********************************************
:: Compile and link test. cpp to executable
cl.exe /std:c++latest /MT /Ot /GL /O2 main.cpp

:: /std -- Control standard functions of the ISO C or C++ programming
language.
:: /std:c++latest -- this /std:c++latest option includes all currently
implemented compiler and standard library features proposed for the next
draft standard.

:: /MT -- Use multithreaded C runtime library
:: /Ot -- Favor Fast Code
:: /GL -- Global program optimization
:: /O2 -- This option specifies options that produce the fastest code in
most cases.
echo **************COMPILING/LINK IS
FINISHED**********************************************************
```

C++23 - Language Features

1. New Rule for Identifiers

Identifiers for alphabets can now use not only Basic Latin 1 symbols but (informally) also "*anything*" that looks like a letter from UCS.

2. Compilers have to Support UTF-8 Source Files Encoding

For C++ 2023, any compiler is required to accept source files encoded in UTF-8. For the MSVC compiler, you may need to specify the compile option /utf8 manually. For the GCC compiler, you may need to use the command-line option: -finput-charset=UTF-8 manually.

3. Side Effects are Still Possible on the Right-Hand Side of the Assignment

The C++17 standard added the rule that all side effects of the right side of an assignment are fully committed before evaluating the left side and the actual assignment.

This feature is still in C++23.

4. Suffix for size_t

C++23 introduces a literal suffix for type std::size_t in the form of using a suffix consisting of both `u` or `U` and `z` or `Z`. Possible suffixies: `uz`, `uZ`, `Uz`, `UZ`, `zu`, `zU`, `Zu`, `ZU`. Short examples: `42uz` and `42UZ`.

Example:

```
#include <iostream>
#include <stddef.h>

int main()
{
#if __cpp_size_t_suffix >= 202011L
    std::size_t a = 1uz;
    std::cout << " size_t uz suffix is supported\n";
#else
    std::size_t a = 2;
    std::cout << " size_t uz suffix is NOT supported\n";
#endif

    return 0;
}
```

Feature Test Macro: __cpp_size_t_suffix

> Warning: MSVC 2022 17.9.0 does not support it.

Documentation:

- https://en.cppreference.com/w/cpp/language/integer_literal
- https://en.cppreference.com/w/cpp/types/size_t

5. Multidimensional Subscript Operator

Another new feature in C++23 is that overloaded subscript operators can take any number of input parameters. A multidimensional subscript operator is particularly useful if your class models some higher-dimensional data collection, such as a matrix.

Example:

```
#include <iostream>

#if __cpp_multidimensional_subscript >= 202110L

class Matrix
{
public:
    int operator[] (size_t i, size_t j) noexcept
    {
        return i*10 + j;
    }
};

int main()
{
    Matrix m;
    std::cout << m[1, 9] << "\n";
    std::cout << "Hello" << "\n";
    std::cout << "Multidimensional subscript operator is supported";

    return 0;
}
#else

int main()
{
    std::cout << "Multidimensional subscript operator is not supported";
}
#endif
```

> Warning: MSVC 2022 17.9.0 does not support it, but GCC 13.1 supports it.

Feature-test macro: __cpp_multidimensional_subscript

Documentation: https://en.cppreference.com/w/cpp/language/operators#Array_subscript_operator

6. Assume Attribute

With C++23, there is a new syntax for telling the compiler about different assumptions about the code in the format `[[assume(expression)]]`.

Example:

```
int div_by_2(int x)
{
  [[assume(x>=0)]];
  return x/2;
}
```

Attribute test-macro: __has_cpp_attribute(assume)

Documentation: cpp reference assume

Analogs:

- MSVC: __assume(expr)
- Clang: __builtin_assume(expr)
- GCC: `if (expr) {} else{ __builtin_unreachable(); }`

7. Withdraw from Optional Garbage Collection

Back in C++11, the Garbage Collection (GC) was added into the Language. But:

- Nobody knows about it
- Nobody uses it

In C++23, the GC has been completely removed from the language.

8. Extended Width Float Point Types

While single and double-precision floating point numbers are by far the most commonly used, they are not the only format in 2010 - 2024.

Today, computer hardware capable of processing 16-bit floating-point data is becoming more widely available. The C++23 standard added the possibility to support some alternative floating point formats.

Specifically, C++23 introduces the following extended floating-point types from IEEE 754:

- `fp16` (std::float16_t)
- `fp32`(std::float32_t)
- `fp64`(std::float64_t)
- `fp128`(std::float128_t).
- In addition, it introduces `bf16` (std::bfloat16_t) the format developed by Google Brain.

In addition to these types, C++23 also provides corresponding suffixes for literals of those types (`[f|F][16|32|64|128]`) and `bf16`, `BF16`. Support for all these formats is optional. Example:

```
#include <stdfloat>
#include <iostream>

int main()
{
    std::bfloat16_t var1 = 1.0bf16;
    std::float16_t var2 = 1.0f16;
    std::float32_t var3 = 1.0f32;
    std::float64_t var4 = 1.0f64;
    std::float128_t var5 = 1.0f128;

    std::cout << "Fixed width FP types in C++23\n";
    std::cout << " Number of bytes for BF16 /item " << sizeof(var1) << " Bytes\n";
```

```
    std::cout << " Number of bytes for FP16 /item " << sizeof(var2) << " Bytes\n";
    std::cout << " Number of bytes for FP32 /item " << sizeof(var3) << " Bytes\n";
    std::cout << " Number of bytes for FP64 /item " << sizeof(var4) << " Bytes\n";
    std::cout << " Number of bytes for FP128/item " << sizeof(var5) << " Bytes\n";

    return 0;
}
```

> Warning: MSVC 2022 17.9.0 does not support it

> Warning: Right now, there is limited support for this new Fixed-width Float-Point type. One compiler where it is available is GCC 13.1 (GCC 13.1 was released on April 26, 2023).

Documentation:

- https://en.cppreference.com/w/cpp/types/floating-point

9. Explicit Object Parameters in Class Methods

C++23 introduces a new syntax that allows you to name the implicit `this` pointer as an explicit parameter.

> For instance, it can be useful during porting Python scripts.

Requirements by C++2023 standard:

- The explicit object parameter must be the first parameter.
- An explicit object parameter, such as self in the example below, is prefixed with the keyword `this` (`this Test& self`).

Example:

```
#include <iostream>

class Test
{
public:
    void f()
    {
        std::cout << __func__ << " [1] this address: " << this << std::endl;
    }

    void g(this Test& self)
    {
        std::cout << __func__ << " [2] this address: " << &self << std::endl;
    }
};

int main()
{
    Test t;
    t.f();
    t.g();
```

```
    return 0;
}
```

In the case of using explicit object parameters in class methods, the `this` pointer will not be available.

Feature-test macro: __cpp_explicit_this_parameter

Documentation: https://en.cppreference.com/w/cpp/language/member_functions#Explicit_object_parameter

10. Ref-Qualified methods (from C++11) and more about explicit Object Parameter (from C++23)

Ordinary class member functions can be called on both live objects (a.k.a. lvalue and const lvalue) and temporary objects (a.k.a. rvalue).

It is possible to explicitly specify what kind of instances a certain member function will be called in these two cases. This is done by adding a ref-qualifier to the member function.

If a member function can only be called on non-temporary instances, a `&` qualifier is added to the member signature at the end.

If a member function can only be called on temporary instances, a `&&` is added to the member signature at the end.

Example:

```
#include <print>
#include <string>
#include <utility>

class Value
{
public:

    void g(this Value& self) {
        std::print("g() for LValue\n");
    }

    void g(this Value&& self) {
        std::print("g() for RValue\n");
    }

    const int& value() const& {
        std::print("const int& value() const&\n");
        return m_value;
    }

    int&& value() &&
    {
        std::print("int&& value()&& \n");
```

```
        return std::move(m_value);
    }

private:
    int m_value;
};

int main()
{
    Value v;

    v.value();
    std::move(v).value();

    v.g();
    std::move(v).g();

    return 0;
}
```

Feature-test macro: __cpp_ref_qualifiers

Documentation: https://en.cppreference.com/w/cpp/language/function#Function_declaration

This example demonstrates that if somebody finds ref-qualifiers too messy for implementing `value()` methods, the C++23 feature with explicit object parameters provides an alternative syntax to write methods `g()`.

Essentially, the new syntax allows you to rewrite the ref-qualified member functions using a slightly different syntax.

11. If consteval - shorthand for std::is_constant_evaluated() in constexpr functions

The constexpr function's body can be executed both in compile time and in runtime. The checking that the current function context is compiled for compile-time can be attained since C++20 via std::is_constant_evaluated(). The C++23 introduced essentially a shorthand for `if (std::is_constant_evaluated())` as `if consteval`.

Example:

```
#include <iostream>

constexpr const char* str1(int)
{
    if consteval
    {
        return "compile-time";
    }
    else
    {
```

```
        return "runtime";
    }
}

constexpr const char* str2(int)
{
    if (std::is_constant_evaluated())
    {
        return "compile-time";
    }
    else
    {
        return "runtime";
    }
}

int main()
{
    const char* str = str2(1);
    std::cout << str;
    return 0;
}
```

Feature-test macro: __cpp_lib_is_constant_evaluated

Documentation:

- https://en.cppreference.com/w/cpp/types/is_constant_evaluated
- https://en.cppreference.com/w/cpp/language/if

C++23 - Preprocessors Features

1. elifdef

The `#elifdef` is the equivalent to `#elif defined(...)`.

Documentation: https://en.cppreference.com/w/cpp/preprocessor/conditional

Compiler Support: https://en.cppreference.com/w/cpp/compiler_support

2. elifndef

The `#elifndef` is the equivalent to `#elif !defined(...)`.

Documentation: https://en.cppreference.com/w/cpp/preprocessor/conditional

Compiler Support: https://en.cppreference.com/w/cpp/compiler_support

3. warning

Shows the given compile-time warning message without affecting the validity of the program.

```
```

```
#warning "Hello"
```

Documentation: https://en.cppreference.com/w/cpp/preprocessor/error

C++23 - Library Features

1. Print and Println

Motivation with a new way to `print` is to work with another API for file stream I/O.

- It is shorter
- There are promises from standardization community that `std::print()` and `std::println()` slightly faster compare to `std::cout << std::format("Hello {}", name);` and faster compare to `printf()`.

Example:

```
#include <print>

int main()
{
#if __cpp_lib_print >= 202207L
    std::println("Hello {0}", 123);
    std::print("{0} ", "Hello");
    std::print("{0} \n", "without a new line");
#endif
    return 0;
}
```

In addition, it's worthwhile to mention that by default, `print()` and `println()` print to the standard output. It's possible to print to another stream, such as std::cerr, as follows:

```
#include <print>
#include <iostream>

int main()
{
#if __cpp_lib_print >= 202207L
    std::println(std::cerr, "Hello {0}", 123);
    std::print(std::cerr, "{0} ", "Hello");
    std::print(std::cerr, "{0} \n", "without new line");
#endif
    return 0;
}
```

From C++23, the format string for format(), print(), and println() must be a compile-time constant so that the compiler can check syntax errors in the format string.

Example:

```
#include <print>
#include <string>

int main()
{
#if __cpp_lib_print >= 202207L

    std::println("Hello {0} {1}", 123, 456);

    // COMPILE-TIME ERROR -- format string is not a constant
    // std::println(std::string("Hello {0} {1}"), 123);

    // COMPILE-TIME ERROR -- format string has two args
    //std::println("Hello {0} {1}", 123);

#endif
    return 0;
}
```

Documentation:

- https://en.cppreference.com/w/cpp/io/print
- https://en.cppreference.com/w/cpp/io/println

2. Stack Traces

The std::exception is an exception that, by design, tells you what went wrong (through the aptly named what() function).

However, what a std::exception cannot tell you is where things went wrong.

Before C++23, there was no standard way for you to find out where in the code the exception was thrown. This limitation is not present in Java or C#, but for C++, it is the case. In these languages (Java and C#), the exceptions carry the addition of a stack trace.

From C++23, you have access to this feature by leveraging the std::stacktrace object in your exception object.

Canonical Example:

```
#include <exception>
#include <string_view>
#include <stacktrace>
#include <print>

class TracingException : public std::exception
{
public:
    TracingException(std::string_view message,
```

```
        std::stacktrace trace = std::stacktrace::current())
        : m_message{ message }
        , m_trace{ trace }
    {}

    const char* what() const noexcept override
    {
        return m_message.c_str();
    }

    const std::stacktrace& where() const noexcept
    {
        // The trace object, which holds tracing information
        return m_trace;
    }

private:
    std::string m_message;
    std::stacktrace m_trace;
};

//__forceinline
void f2()
{
    throw TracingException("Ouch");
}

//__forceinline
void f1()
{
    f2();

}

int main()
{
    try
    {
        f1();
    }
    catch (const TracingException& ex)
    {
        std::println("Exception Type {0} caught: {1}; trace: \n{2}",
typeid(ex).name(),

ex.what(),

ex.where());
    }
}
```

Feature-test macro: __cpp_lib_stacktrace

Documentation:

https://en.cppreference.com/w/cpp/header/stacktrace

https://en.cppreference.com/w/cpp/utility/basic_stacktrace

3. Flat Associative Containers std::flat_set and std::flat_map

Typically, a `std::set` and `std::map` are implemented by balanced red-black trees. The `flat_set` is simply backed by an ordered sequential container. As a consequence:

- Flat sets consume less memory.
- They provide faster element lookup.
- They provide faster iteration than tree-based sets.
- They interplay nicely with memory locality principles.

On the other hand, tree-based sets are better at inserting and erasing elements.

Similar to `std::flat_set`, there is `std::flat_map`.

This new container has the same API as `std::set` and `std::map`.

> Warning: At the moment March-2024 no compiler supports it:
> https://en.cppreference.com/w/cpp/compiler_support

Feature-test macro: __cpp_lib_flat_set, __cpp_lib_flat_map

Documentation:

- https://en.cppreference.com/w/cpp/header/flat_set
- https://en.cppreference.com/w/cpp/container/flat_map

4. Compiler Hint std::unreachable()

Marks that the current scope or place is unreachable. If the code comes to this point, then the behavior is undefined. The point of this call is to tell the compiler that part of the code does not need to be generated. A canonical example for the use case is a switch statement, which can be designed and dispatched by all `cases`.

Example:

```
#include <iostream>
#include <utility>

bool decodeBit(int x)
{
    switch(x)
    {
    case 0:
        return false;
    case 1:
        return true;
    default:
```

```
            std::unreachable();
        }
    }
    int main()
    {
        std::cout << decodeBit(1);
    }
```

Feature-test macro: __cpp_lib_unreachable

Documentation: https://en.cppreference.com/w/cpp/utility/unreachable

Analogs:

- MSVC: __assume(false)
- GCC: __builtin_unreachable()
- CLANG: __builtin_unreachable()

5. Multidimensional View std::mdspan

The `std::mdspan` is a multidimensional view of a contiguous sequence of objects interpretable as a multidimensional array. This is a multidimensional extension of std::span from C++2020.

Feature Test Macro: __cpp_lib_mdspan

Documentation: https://en.cppreference.com/w/cpp/container/mdspan

6. Stream API with Raw Buffers via std::ispanstream

In some circumstances, you may want to use the stream for I/O, but you want to eliminate the need to copy around different places.

Unfortunately, standard streams such asstd::stringstream from `sstream` have internal buffers. And dealing with them via configuring rdbuf is, firstly, pretty low-level. And, secondly, there is no guarantee that stream classes do not use their internal buffer.

The C++23 introduces `std::ispanstream` which provides a way to use a streaming interface on top of the raw buffer.

Feature-test macro: __cpp_lib_spanstream

Documentation: https://en.cppreference.com/w/cpp/io/basic_ispanstream

---

How to cite this C++ Technical Note

In case you decide to cite this note, please use the following BibTeX entry type:

```
@techreport{burlachenkok2020cpp,
author = {Burlachenko, Konstantin},
```

```
title = {C++ from 1998 to 2020},
year = {2022},
institution = {GitHub},
url = {https://github.com/burlachenkok/CPP_from_1998_to_2020},
note = {Accessed: 2022-01-14}
}
```