Technical Note. From C++1998 to C++2020

Konstantin Burlachenko

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

Correspondence to: konstantin.burlachenko@kaust.edu.sa

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: Sep 09, 2022

Table of Content

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, and C++20. If you do not know C/C++, this note is less likely for you because it contains subtle technical details for people who are at least familiar with it a bit. 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/C++.

Do not get us wrong. If you have never seen the C/C++ language to obtain knowledge, we recommend first dedicating some time to reading original books by Bjarne Stroustrup. It would be only 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 for whom that language is new firstly read any of that 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/C++.

Sometimes you need to write software to 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 reasons. In such circumstances, the defacto standard can be another programming language: not C/C++ at all or dialect of C/C++. Analyzing when it is good or bad is out of the scope of that technical note.

That note is mainly based on materials from the Reference 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 language constructions of C++
- Obtain a pretty in-depth overview of new features from C++11/14/17/20
- 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*, but we don't want to have any of the extremes of both types.

Glossary

C/C++. By C/C++, we mean C or C++ programming 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.

A Deep Copy. A deep copy copies all dynamic memory objects referred to by any pointer members.

Upcast. Casting object to its base class

Downcast. Casting object to the derived class.

Function Signature. The combination of the function name and the parameter list is called the signature of a function.

Function Prototype (function declaration). The language statement that describes a function sufficiently 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/function instantiation.

Function Object (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 virtual function is to enable the derived class versions of the function to be called polymorphically. The purpose of a pure virtual function is have polymorphically bahaviour in 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 with an address in memory where you can store something. Informally that is something to the left of the operator equals.

RValue (expression). An RValue evaluates a result that is stored only transiently. An expression from which the address cannot be taken. Also, this is something that, at least in principle, can be encoded in the code of generated instructions for the processor. In 99% of cases, these are unnamed temporary variables. But a good counterexample of something that is an *RValue* but has the name `this.`

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

LValue Reference (for all C++). Typically, an LValue reference is an alias for another variable. Lvalue object may be bound to the LValue reference through syntax:

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

RValue Reference (only for C++98/03). In C++03, it is a usual const regular reference to a temporary object or expression that can be used from the right-hand side of the operator `=`.

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 soon will be deleted (xvalue expression)
- Explicitly unconditionally casted reference to the object through `std::move` to an RValue reference. The `std::move` after moving, brings object for moving is applied to a valid but undefined state.

> Reusing an object after moving from is *legal* and *valid*. In one of the talks in CppCon Nicolai M. Josuttis member of C++ Standard Committee, explicitly highlighted it. In that case, you should reinitialize the object using class API or the logic behind the class.
>
> What was known in C++03/98 as *RValue Reference* 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 more hard case is to identify tokens when there are no whitespaces.

Pure Function. The pure function is type of function (used in case of using `constexpr`) in C++ when function implementation make C++ function conincident with matematical function. Specifically we can name function as pure if the following holds:

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

Incomplete Type. In C and in C++ there are two cases when you can use a type with a not yet undefined size - creating a pointer for a type and creating an alias name via a *typedef* or *using* Keywords. Essentially it is possible because the size is not required for a such declaration of a pointer and alias name. [2, p.150]

```
class A;

typedef A B;
using BB = A;
A* ptr;
```

Motivation

The C/C++ programming language represents a pretty thin abstraction over the underlying hardware. The software level below C/C++ is Assembly Language for your computing device. Why computing 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.

Nowadays, in 2022 due to Tobex Index July 2022, the interpretable programming language Python is the most popular in that world. From the graphics, you can observe that Python is slightly beyond C in terms of popularity. Interestingly, Python has been designed originally only as a replacement for Bash. That has been described in that Blog Post written by author of Python Programming Language:

> "...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 that today people try to apply Python beyond launching scripts but creating other user space applications. When (a) Underlying Algorithms that you need are implemented in C++ or inside Hardware; (b) They are available via Python bindings; (c) The overhead of Python Interepreter is negligible; (d) There is a big part of the system has already been implemented in Python - It may be a choice to use Python in that case.

We think the main reason popularity of Python is primarily due to the fast learning curve measured by three days (only Language, no 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 in DRAM or any memory/compute inside any device connected to the computer matters, even programming only in user space.

Downsides of Interpretable Languages

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

   > 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 Operation 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 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 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 OS API. If the goal is portability between different OS, more correctly is to solve it via standardization of API to OS. Creating extra software layers, especially in the form of interpreters, 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. In fact, you do not even have enough tools to precisely handle just usual Virtual memory in your process.
5. The interpreter as a computer program adds an extra level of abstraction. The standard implementation Python interpreter is CPython. It is called CPython because it has been implemented in C/C++. Such software as an interpreter improves the time for completing the project from social point of view, but implementation is suboptimal.
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.
7. Uncontrollable memory allocations in a program that should work for a long time and during runtime require extra memory allocation may lead to memory fragmentation and other memory problems.
8. Compiler optimization tricks such as code inlining are out of the scope of any interpretable language because for performing such optimization you should have a compiler. The elimination of the compiler stage will make such optimizations impossible.
9. During creating multithread implementation you should be careful about memory fences, synchronization, data races, atomic operations, absence of storing some objects in registers. In reality, implementation of interpreters is typically highly leveraged into existing C/C++ libraries because creating such modules of functionality in an interpreter by itself is not effective enough. But it is not true that all C/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 really 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 is not so easy).
10. Garbage Collector (GC) brings various limitations to any programming language. For example, GC disallows any pointer arithmetics. (For details, please look at Lecture 11 from 6-172. Performance Engineering of Software Systems at MIT).
11. Due to high abstraction, Interpretable Languages violate memory locality principles because almost every object is allocated on the heap. 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.
12. Processors have a limited number of registers. If you have too many objects with too many wrappers around them the useful load for a real final compute device is smaller and degrades.
13. There is no way to use special registers or special instructions of the processor from typical interpretable language (Bash, Python)

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/C++

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

2. Powerful expressivity of C++ is a technical power. At the same time, it's its weakness for 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. And C++ is used only when necessary (E.g., you need to work with the hardware directly; you need to have algorithms/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 typically user space applications.
5. C++, to some extent, forces you to be aware of 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.
    - On another side, 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.

People continue to predict that C++ will die. It's an ongoing three decades 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 and more complex.

Deep Principles of the Language

The Language started as a project in Bell Labs in 1979 ([3]). The principles of C++ language, which B.Stroustrup put into the Language, were documented between 1981 and 1991. They existed even before the decision of standardization that took place in 1989. More importantly, principles of the Language, even in 2022, are still inside it, and they are the heart of the Language ([3]):

1. No implicit violation of static type system.
2. Provide good support for user-defined types similar to built-in types.
3. The locality of memory access patterns for variables and arrays is suitable for hardware in the long term.
4. Zero-Overhead principle:
    - a. What you don't use, you should not pay for.
    - b. If something is built-in in the Language, 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 against precisely defined interfaces and 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 can implement the algorithm in Python:

- Algorithm is short and suitable for CPU.
- Library does not exist, or Library exists but does not provide Python bindings.
- Library does not provide enough configuration
- You need to change something fundamental inside C++ Library, and you don't know C++.

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
- C++ program does not use any special optimization techniques. It's usual C++ code.

The test compares 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/C++ arrays used in C/C++.

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 with 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/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/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 language as Cython), it's not easy to be better even 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/C++, and finally, the code is compiled as 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 to a situation where you are not using an interpreter anymore. Nevertheless, It can be helpful to be aware of such a possibility.

Standards for the Language

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

Scott Meyers gave a presentation about C in YANDEX 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** - no leaks and standard libraries supported.
2. **Strong guarantees** - operation is fulfilled completely or not.

All containers in C++98 provide a fundamental guarantee. And some operations (for example, `std::vector<T>::push_back`) give a strong guarantee.

Stages of Source Code Translation in C++

A source code for C/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 broken into lines.
2. Processing trigrams. All available C trigrams can be obtained from [2,p.15].

```
#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 mainly like the tokens used by the C compiler, except for some differences. Example: Token `##` is a concatenation operator for words in a source code. It is the valid operator for the C preprocessor, but it is an invalid token for the operator in a proper C/C++ program.
6. Processing the program code by the preprocessor. The preprocessor can be built-in into the compiler, or it can be an independent program. For details about available preprocessor language, please read [2, p.43] or documentation for any de-facto standard toolchain like GCC.

The output of preprocessing of the source file is named as *preprocessed source* and typically has the extension `*.i`. For example, obtaining such a source file from clang can be achieved via `clang -E.`

The Compiler and Linker. Briefly

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

The final binary format (ELF for Linux and PE for Windows) is also not under the obligation of creators of Language or userspace developers. It's under the responsibility of the creators of the Operation System. There are situations when the target device in which the program will be executed has no operating system. The case of launching program on target device with no Operation System sometimes is 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 below if you are curious about how a compiler compiles source code. It's possible to be productive even without the knowledge below especially during creating only userspace 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.

Lexical Analysis

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

An important part is that the C/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;
// invalid tokenization: tokens (b, --, a)
c = b--a;
// valid tokenization:tokens(b, -, -, a)
// but C/C++ compiler does not do that
c = b - -a;
```

The concept of whitespace in C/C++ includes different keyboard spaces and comments. In C/C++ and most programming languages, the tokens fundamentally can be one of the following types:

- a. Operators

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

After finishing, the Lexical analysis, the program consists of a sequence of tokens.

Syntax Analysis

The compiler is based on the language rules typically described by Backus–Naur forms for Context-Free-Grammars (CFG). The Grammars by themselves are studied in a mathematic area called *Formal Languages and Grammars Theory*. That area of mathematics is essential for Compiler 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 corresponding Language Standards.

Semantic Analysis

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 that analysis, the Semantic analyzer inside the compiler is used.

Code Optimization

At that moment, we have constructed AST for a program and augmented it with information from the semantic analysis stage. At that moment, we can traverse AST and translate that code into more low-level construction expressed as Assembly Language or Intermediate Representation (IR).

Before that stage, the stage of code Optimization is occurred. Compilers' innovations based on various fields of science and engineering that mainly bring considerable speedup are exploited in that 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. Keys run in a predetermined order that usually seems to work well. Some examples of optimization technics that happens at that moment:

- Convert one arithmetic operation into more cheap operations via using bit tricks and logic/arithmetic shifts.
- Replace stack allocation storage with storing variables in the processor's register.
- Optimization for 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 pointer/reference of an object of the structure/class type.
- Remove dead-end code never executed in the program's control flow across as compiled source files.
- Function inlining. The compiler uses its heuristics to decide what to inline and what to not. Sometimes there is a toolchain extension that forces that.
- In case of using global program optimization compiler and linker jointly may want to try inline even function definition from another compilation unit.
- Remove Hoisting (also known as loop-invariant code). Try to remove recomputing loop-invariant code inside the loop.
- Vectorization. Leverage into vector registers (like SSE2, Arm Neon) when possible.
- Loops optimization: unrolling; loop fusion (also known as jamming) to combine multiple loops over the same index range; eliminating wasted iterations in the loop.
- Figure out with Memory Aliasing and apply optimization for non-aliased pointed expression. For details please check Compute Optimization Relative Information.

Various things of optimization are out of the scope of the compiler. And can only be solved by the creator of the Algorithm/Method. Even we think there is a possibility of research to provide that information 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 what exactly compilers emit:

1. Compiler emits the final binary code for target Instruction Set Architecture (ISA). (Example: Microsoft Visual C compiler do that.)
2. Compiler emits the program text written in Assembly. But in fact the process of producing final binary code is under responsibility of Assembler program. You can obtain such assembly source from preprocessed file manually for clang toolchain via invocation of `clang <source_file.i> -S -o -`.
3. With the coming LLVM project there is in fact intermediate layer between High Level Language(C++) and ASM for target ISA. That layer is called Intermediate Representation (IR). And contains tree stages conversion:
    - At the *first stage* the input preprocessed code is converted into 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 that 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 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 a big programs. ASM language has a close relation to target compute 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/C++ can be emitted (materialized or generated) into different instructions in 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 Assembler is saved into *object files*.

The Assembly code by itself obey Instruction Set Architecture *ISA*. The *ISA* specifies instructions, register, 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). The particular implementation of ISA is called Microarchitecture in Electrical Engineering(EE) terminology. Different vendors can provide the support of the same ISA, but Microarchitecture is typically under NDA. The Microarchitecture is the lowest level of computation and it's under the responsibility of Electrical Engineers, not Computer Science people.

There are online tools such as [11] Compiler Explorer that allows demonstrate Assembly code during using various compilers for C++ online and can be very usefull for educational purposes and via using color show correspondence between C++ code and Assembly code.

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/C++ entities), using specially provided flags, perform a whole-program/global program optimization or optimization specified via command links.

The nuances of compiler/linker organization are out of the scope of C++ language and can vary from vendor to vendor. For example, for GCC, the Assembler is a separate program from the C compiler physically. In another toolchain, e.g., from Microsoft Visual C compiler, the translation to final binary code is inside their C compiler.

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

What is Impossible Even in C/C++

- *Address individual bits*. Few machines can directly address an individual bit. Even if the device allows it, it is out of the scope of C/C++, to be honest. Of course, you can operate on bits, but not directly.
- *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 and are only a bit familiar with C++, we would like to enumerate several not complicated things for you.

1. The logical operators `&&` and `||` is called short-circuit evaluation due to their behavior of evaluating subexpression only when needed for the logic expression evaluation. However, the bitwise operators `&` and `|` do not short-circuit.
2. The difference between the two pointers at the level of the C/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`. The potential confusion here 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. Substrings are always specified using beginning index and length, not beginning and ending indexes as in some other languages. Keep this in mind when migrating from other languages.
6. Never return the address of an automatic, stack-allocated local variable from a function.
7. When a class member function executes, it 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`. A static member function is not associated with any class object, it has no `this` pointer, and `const` property is not applied to them.

10. The primary purpose of operator overloading is to increase the ease of writing and the readability of code that uses your class.
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 calls a base class constructor. If a user-defined derived class constructor
    does not explicitly call a base constructor in its initialization list, the default constructor will be called.
13. A table of virtual function pointers is created for each
    class that contains virtual functions. The only time you should even debate whether the overhead of a virtual function table pointer is worth 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 names should not be used by non-compiler and non-STL writers both for C and C++ languages. With that rule for a long time, people escape conflicts between naming of compiler-specific entities and entities of the construct program.

> Sometimes compiler writers violate that principle.
> For example, The Linux Programmer's Guide mentions standard predefined macro uses `unix` and `linux` during compilation for the Linux platform. That special names are not the names that follow C/C++ conventions to distinguish names used in the program and for compiler writers.

> 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 `__.` That special names are not the names that followed that rules.

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

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

17. In C/C++ unary operators have higher priority than binary operators.
18. Operators `==`, `!=` have higher priority than logical connectives.
19. The C++ standard guarantees that the life of a temporary object if it is **LValue** (i.e., that temporary object 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.
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.

About C/C++ Preprocessor

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

Preprocessor macro extensions in C/C++ have the following property. Once an extension replaces a macro call, the macro call search process starts from the beginning of the expanded extension for further replacement.

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

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

For the C/C++ preprocessor, any undefined identifiers that appear after the conditional directives `#if` and `#elif` is 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 an enumeration order of include paths varies between compiler toolchains, so you may figure it out for a particular toolchain by experiment.

Include Files Naming

1. 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 headers files expose different behavior in terms of using their names from the global namespace.
2. Standard headers with naming as `<X.h>` define function names in the `std` namespace and also **import those names into the global namespace**.
3. Standard headers with naming as `<cX>` defines function names only in the `std` namespace. ([1], 9.2.2, page 247).

Predefined Identifiers and Macros

`__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 the argument. When the macro is invoked, all the tokens in its argument list `...`, including any commas, become the variable argument.

`__cplusplus` - C++ version

`__STDC_VERERSION__` - version of standard C.

Example of checking the version of C/C++ compiler mostly 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/C++ and 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).

Next, 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`, `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/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`
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`	

Suffix `s` has been made available since C++17, and other suffixes were always in any version of C/C++.

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. For Windows it's UTF-16, for Linux it's UTF-32.
2	`u'a'`	ucs2 symbol. Pretty like UTF-16, but surrogate pairs are not supported in UCS-2.
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. Not all letters in Unicode can be represented by a single 8 bit character for UTF-8 or single 16 bit character for UTF-16.

Function Call Nuances

1. There are only two kinds of function and function calls in C++:
   - *Ordinary function call.*
     A static member function is an ordinary function ([6], 9.4).
   - *Member function call*.
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; **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++". To link functions in C++ style:

   ```
   extern "C++" void f()
   ```

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 when you modify the value more than once is a bad practice. But in fact, 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 legal 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) There can be more than one definition of the following things in different translation units (cpp source files):

1. class type
2. enumeration type
3. inline function with external linkage
4. class template
5. non-static function template
6. static data member of a class template
7. member function of a class template
8. template specialization for which some template parameters are not specified

That same definitions are acceptable when:

1. They are in different translation units
2. They 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 but is assigned to the 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, there are three allowable implementations for signed integers consisting of n bits. For details, we recommend looking at [2, p.125], but short 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 positive number representation plus 1. In Assembly for x86 that 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 zero are possible. As implication 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.

The most famous representation for signed integers by hardware vendors is *two's complement* notation.

Starting from `C++20`, there is only one signed integer representation, and it's 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, then `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, that underlying type for C++ has some underlying integer type to store values, but it has not been specified.

Starting from C++11 now we have two type of `enum`:

- Strongly typed enums(or scoped enumerations)
- Usual enums (or unscoped enumerations).

Strongly typed `enum` do not support implicit conversion to `int`. It's possible to specify for them the underlying type. The underlying default type for strongly typed `enum` is `int`. Example:

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

For ordinary `enum` in C++ 11, as in C++ 98, nothing is said about the underlying type by default. If the underlying type is specified, you can make a forward declaration for `enum`.

For usual `enum` starting from C++ 11, it's possible to specify explicitly the underlying type.

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

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, as it were, through the namespace of a new type. Starting from C++11 for `enum` of known size may be forward-declared.

Integer Types Nuances

1. The compiler's use of `unsigned` or `signed` in the absence of an explicit type specification is not defined for `char`.
2. The compiler's use `unsigned` or `signed` in the absence of an explicit type specification is not defined for bit fields.
3. In general, the result of doing the right shift for signed integer types in the case of negative numbers is undefined before C++20. The compiler implementer can perform either a logical right shift or an arithmetic shift ([2], p. 252). **Since C++20 right-shift on signed integral types is an arithmetic right shift, which performs sign-extension.**

Auto Type Deduction

The `auto` in C++98/03 was an explicit memory type for local objects, but starting from C++11, it's a type automatically deduced similar to `template` arguments.

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 `auto&` or `const auto&` [4, p.309].

The `auto` works in usual functions for variables and for arrays. For auto, the deduction is identical to the output for `template` function arguments.

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

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

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

`auto` can be used jointly with constant volatile type qualifiers (cv) and with reference and pointers. Examples:

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

The close-by conception is decltype. It provides the ability to derive the type of expression without evaluating it. There are some subtleties with `decltype`. In fact, `decltype(x)` and `decltype((x))` are often different types. If the argument is an unparenthesized expression or an unparenthesized class member access expression, then decltype yields the type of the entity named by this expression. The inner parentheses cause the statement to be evaluated as an expression.

```
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 `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 `for` loop named as *"range based for loop"*. Example:

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

Range-Based for Loops valid for any type supporting the notion of a range. One of the following constructions should be valid:

- `obj.begin()` and `obj.end()`
- `begin(obj)` and `end(obj)`

Technical Differences between C and C++

Many times in the past, people aware of C/C++ talked that C++ and C are different. Let's take a look at what it means concretely. All 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 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: Tradition 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++.

7. Although for C++ type tag names (struct, union, `enum`) are implicitly declared using `typedef`, they can still be hidden by variables in the same scope `s s;`.
8. C99 has pointer qualifier `restrict`, which is not in the official specification of C++98/03/.
9. Support of **flexible array type** array. In C99 such 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 that case, the size of the structure's element *d* is omitted. 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 a support of `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 concept allow in a portable way perform varying allocation 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

```
char a[3]="12"; char aEquiv[]="12";  // Ok in C/C++
char a[3]="123";                     // Ok in C, but not in C++
```

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 definition of `struct` and `union` in C++ have block scope.

14. `const` declarations are `static` by default in C++, but `extern` 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."* It also follows from this paragraph that declarations of non-const variables declared on namespace level have extern linkage by default in C++.

15. C++ declaration `void f()` is equivalent to void `f(void)` in C. The declaration in C `void f()` state that 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. Also, in C99, component-wise initialization is allowed, which is not permitted in C++98/03.

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. And 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 initialization of them.

21. In C++, there are more stringent requirements for inline functions - an `inline` function must be declared as such 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, 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 to actually what to do:
    - inline calls
    - not inline
    - partially inline.

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

25. There is no such type as a reference in C. (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 with namespace, 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 are not called.

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 identifier was absent in C++98/03.

33. In C++, operators not presented in C language usually have the highest precedence, 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.

35. Before C99, i.e. in C89, C89 with the extension, there was a restriction on where automatic (stack) variables can be defined - only at the beginning of a local block. In C++98/03 already, you can declare local variables anywhere.

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/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 applied for 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/C++, a function pointer expression can be used to call a function without explicitly dereferencing the pointer, i.e., you can call the function via 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 include 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 character is specified in the `CHAR_BIT` macro in C Language. All objects of the same type by C/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 the 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 to some object/data is a pointer. To store (or serialize the value of 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 formally not to a function.
6. A special value in C/C++ called a null pointer equal to a null pointer constant. A null pointer can be converted to any other type of pointer.
7. A null pointer in C/C++ is:
    - An integer expression that yields zero.
    - Or an integer expression casted into a pointer.

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

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

In C++11, in addition to NULL, you can use `nullptr`. That keyword stands for 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/C++ of an identical physical mapping of components "from this
beginning".

9. In C and C++ there are 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 what endian the computer has where the program will
run.

10. Structs are not allowed to perform comparisons with `==` or with `>`. The
fundamental nature of this restriction in C/C++ is because, 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` is ignored.
    It's due 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."

    It also follows from this paragraph that declarations of non-const variables
declared on namespace level have `extern` linkage by default in C++.

12. A compile-time string literal in C/C++ is statically allocated so that it is safe
to return one 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

For Visual Studio compiler:
1. `new(std::nothrow)` - does not throw an exception
2. regular `new` - throws an exception. You can customize the behavior using linker
options: http://msdn.microsoft.com/en-us/library/kftdy56f.aspx
```

```
## Placement New

There are such variations of the new operator:

1. Usual 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:

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

2. Overloaded operator new as a new global function. 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 calculated automatically via `sizeof`. After that, 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 actually 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 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, array of non-POD, or a reference.

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

- No virtual functions
- No virtual base
- Zero or more base classes of standard-layout class types
- No two base classes of the same type
- All data members should have the same access control
- All data members are defined in the most base class or most derived class.
- No restriction for static member functions and static members

Built-in Type Conversion

Before going into technical details about type conversion, let me be honest - it's hard to remember them, so possibly it is 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 technical details.

Prohibited Conversions

1. Converting a pointer to a function, a pointer to a data, and the other side direction is not allowed in C/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 type as well.

4. In C, implicit conversion from integer to enumerated types is allowed because in C, `enum` is synonym of `int`. In C++ it is prohibited.
5. In C and C ++, implicit conversion from enumerated types to integer types is allowed.
6. Converting a pointer to a function to a pointer to data and the other side is not allowed in C++.

The Sequence of Type Conversions Rules in C/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 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 nonzero 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 converting 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 guarantee 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, ordinary unary conversions are performed. The goal is to reduce the number of arithmetic types.

- An array of type T $\rightarrow$ pointer to the first element (not applied for arguments of `operator &` and `sizeof` operators).
- Function $\rightarrow$ function pointer.
- Conversions from an integer type of rank below int $\rightarrow$ to `in`.
- Conversions from unsigned integer types lower than `int`, `int` represent all values $\rightarrow$ values are cast to integers.
- Conversions from unsigned integer types lower than `int`, but `int` does not represent all $\rightarrow$ 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`, `double`, `float`, and the second operand has a rank lower, 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. Unsigned operand and lower-ranked signed operand $\rightarrow$ unsigned type.
18. Unsigned operand and signed type operand of higher rank $\rightarrow$ signed 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. Also, besides that, `float` is always promoted to `double`. The float is not converting 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 disadvantages:

- Waste of time analyzing the assignment of objects to different namespaces
- Various additional nuances:
    - A local variable or a variable declared via `using` hides external variables in relation to the block of visibility.
    - When libraries that declare many names are made available through the `using` directive, it is important to understand that unused name conflicts are not considered errors.
    - 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 features being added to it. The `using` directive applies more to namespaces than to classes.

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 does not pollute the namespace.

```
namespace NameSpace {
  structType{};
  void func(Type x)
  {}
}
...
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 your type is in.

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).

2. A locally declared name and a name declared with a `using` directive hide a non-local declaration.
3. Local name declaration takes precedence over NS name, but 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". It's just global. It differs from all namespaces (including the unnamed one).
   The global namespace differs from those defined through 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.

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 declaration:
   7.1 Creating a synonym for a variable through `using NS::x;` It's called **using declaration**.
   7.2 Creation of synonyms for all variables from namespace - through `using namespace NS;`. It's called **using directive**.
8. Placing 7.2 inside another NS opens up the way to combine/mix features 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 also made available with using declarations i.e. via `using NS::x;` take precedence over names made available via using directives i.e. `using namespace NS;`
11. Namespaces can be nested. To create an alias, you can use a construct like the following:

```
namespaceAA = NameSpace::NameSpace2;
```

12. Namespace Search Rules in case of using nested namespace

Example with a variable "i" in ANSI ISO IEC 14882, C++2003, 3.4.1, (6) (page 30).

```
namespace A {
  namespace N {
    void f();
  }
}
void A::N::f() {
    i = 5;
}
```

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

Some relative things are here: https://en.cppreference.com/w/cpp/language/unqualified_lookup

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

```
// Make cout available without qualification
using std::cout;
// Make all names in std available without qualification
using namespace std;

// Defines Big as a type alias
using Big = unsigned long long;
// Defines BigWithTypedef as an alias for long long (Typedefs in C++20 are obsolete).
typedef unsigned long long BigWithTypedef;
```

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, separating error handling code from "normal" code is a good strategy. Throwing an exception may leave the object in an invalid state.

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 an object. (Escape from a block by throwing an exception cleans up all created local automatic things in reverse order of creation). Writing correct exception-safe code using explicit tries can be a difficult task.

Extra about Exceptions

- You can group exceptions by inheritance relation.
- 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 can be seen for:

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

- 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"**.

> *How memory for exception object is allocated?*
>
> Unfortunately answer for 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."*

- If a destructor called during stack unwinding exits with an exception, 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 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;`.
- When an exception is thrown in a constructor, the object's destructor is not called.
- 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 destructor due to normal call of the destructor or during stack unwinding, you can use: `uncaught_exception()` in the destructor.
- The basic errors classes are `exception`, `logic_error`, `runtime_error`. Some others class: `bad_alloc`, `bad_cast`, `bad_typeid`, `bad_exception`, `out_of_range`, `invalid_argument`, `overflow_error`, `ios_base::failure`.
- If a function tries to throw an exception it didn't declare, it will result in a call to `std::unexpected`, which by default pulls std::terminate (p.429, B. Stroustrup, special edition)

```
intf(); /* 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 functions header, you can be sure that this function will never throw an exception.

Construction of 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. Exact type matching, or the matching achieved by trivial conversion (array name to a pointer, function name to function pointer, type `T` to `const T`).
2. Correspondence is achieved by the promotion of integral types and the promotion of real numbers to integers. (`char` to `int`, `float` to `double`).
3. Correspondence achieved by standard conversions (`int` to `double`, `double` to `int`), pointers to derivatives to pointers to base classes. 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. 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 challenge is considered ambiguous and is rejected.

In context of overloading class member functions there is one problem when we want to take into overload resolution not only members from the current class, but also member functions of a base class. C++ treat differently the class scope of base and derived class in context if function names are identical. Very popular fix the standard behavior of the compiler, not trying to find an overloaded function in B is the following:

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

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

Starting from C++11 `using` keyword is also used for Base Constructor inheritance. Please check section *"Miscellaneous Features of C++11"* of that document.

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 the more specialized of the two, the less specialized is discarded.
3. Overloading allowed for functions from steps 1-2 and 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 is not found.

Resolving the Overloaded Binary Operator for x(op)y

1. If X(type of x) is a class. Find out if the operator is defined as a member of class X or a base class of X.
2. View operator declaration in the context of x(op)y expression.
3. If X is declared in namespace N, look for operator declaration in namespace N.
4. If Y(type of y) is declared in namespace M, look for operator declaration in namespace M.

Operators Overloading Rules in C++

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

According to the Standard, operators cannot be overloaded as static methods of a class. It is possible to overload them:

- as functions
- as non-static class methods. (The operators `=`, `[]`, `->` can be overloaded only in this way).

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 t`ypename` in the argument type declaration for a template class/function/method.

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

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

```
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.

```
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 here: http://stackoverflow.com/questions/610245/where-and-why-do-i-have-to-put-the-template-and-typename-keywords.

Class Constructor and Destructors

Logic behind executing Constructors

The order of working out the 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:

1. Depth-first, left to right constructors of virtually inherited classes whenever they are located in the inheritance tree.

```
#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()
```

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 (again, 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/C++, when using custom conversions via constructors without `explicit` keyword or defined `type conversions` - only one level of implicit conversions is allowed.

Logic behind Executing Destructors

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

1. Working out the body of the destructor for class members
2. Destructor of direct non-virtual base classes

3. 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 behaviour 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 about 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/Suppress 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`. It can be used for change normal accessibility, example:

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

For unsuppressing of implicitly generated special member functions.

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

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

You can disable using (any) function/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)

```
class X {
public:
  // "ordinary constructor": create an object
  X(Sometype);
```

```
  // Default constructor
  X();
  // Copy constructor
  X(const X&);
  // Move constructor
  X(X&&);
  // Copy assignment: clean up target and copy
  X& operator=(const X&);
  // Move assignment: clean up target and move
  X& operator=(X&&);
  // Destructor: clean up
  ~X();
};
```

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};  // bracket initialization
int a {15};           // bracket initialization
int count(4);         // functional notation for initialization
int z{};              // zero initialization

const int k1 {5};      // brace/uniform initialization (form-1)
const int k2 = {15};   // brace/uniform initialization (form-2)

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

2. Static variables are initialized to zero for the corresponding type. If the initialization list is empty, the array and structure variables are initialized to zero.
3. 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.
4. Types of initialization in C++

- zero_initialization
- default_initialization
- value_initializati,on

More about initialization: 8.5, 8.5.1 Initializers from C++2003-10-15.

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 and heap, arrays are impossible to init, 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 to `std::initializer_list<T>`. Ctor. with initialization list argument has priority during overloading.

Type deduction in template functions does not work for `std::initializer_list` type. But for auto 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() {
//  f1({1,2,3}); // DOES NOT COMPILE
  f2({1,2,3});
  auto f3 = {1,2,3};
  return 0;
}
```

The `std::initializer_list` stores initializer values in underlying array has these member functions:

- **size**. Number of elements in the array.
- **begin**. Pointer to first array element.
- **end**. Pointer to 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."*.

Various Constants Flavors

const (C++03)

The keyword `const` prevent `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.

`constexpr` - indicating that it *should be possible* to evaluate the function at compile time if given constant expressions as arguments.

It can be run in compile time and in runtime.

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

C++11 `constexpr` functions had to put everything in a single return statement. C++14 `constexpr` functions, not necessarily had 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, `constexpr` is implicitly thread-safe.

Function can be run in compile time and in runtime. In C++20 there is a way to distinguish between compile time and runtime via using `std::is_constant_evaluated()` inside function with `constexpr` modifier.

The `constexpr` functions requirements:

- Must resolve dependency at compile time.
- Can not have `static` or `thread_local` variables inside
- Function should not have side effects (pure in mathematical sense).
- Have *the potential* to be run at compile time.
- Function with `constexpr` modifier is **pure** function.

consteval (C++20)

The `consteval` expressions generated an immediate function.

It can only be run in compile time.

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

- Must resolve dependency at compile time
- Can not have `static` or `thread_local` variables inside
- Function should not have side effects (pure in mathematical sense)

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 in fact can happen in *compile-time* or in *run-time*.

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 is mutable and is not nessesary a const, but initialization must be performed in compile time.

```
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. 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;
}
```

Allowable Reformulations

In C++ and C, the order in which subexpressions are evaluated is not defined `A(F(), G())`.

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 it mean `A(*B)`? It can be declaring a variable or calling a function. This ambiguity makes the C grammar Context-Sensitive and not LALR(1).

In C++, access control is 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 a type for aligned but uninitialized memory buffers:

```
typedef std::aligned_storage<sizeof(X), alignof(X)>::type
XArrayType;
XArrayType buffer;
```

The `buffer` can hold an X object. Size and alignment are fine.

```
X* x = new (&buffer) X(); // construct X in buffer
```

- Facilitates use of stack-based buffers (buffer is not on the heap)
- Like alloca in POSIX, except standardized, 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/C++, when you deal with pointers, for example, in a function mentioned above, it can be two possibilities:

- Pointers `a` and `b` refer to the exact location in memory, at the start or during runtime. It's legal for such a function signature.
- But maybe pointers `a` and `b` never refer to the exact location.

Without any extra help, the compiler has serious problems deciding what the case is. And be default compiler assume (1) case. However, there is a way to specify pointers as `restrict (In C99)` via the following syntax:

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

The C++ does not have such a keyword even up to C++20, but toolchains typically provide C++ extension via the `__restrict` extension.

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 pointers are the only way to access the object where the pointer is pointed. For functions 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., memory in pointers are pointed is used for the

write. In the case of using objects by `RValue,` the restriction does not impose any semantic restriction of memory not overlapping where pointed objects are located.

Lambda Functions

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

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

- []. The opening square brackets are called the lambda introducer.
- (int x, int y). lambda parameter list between round parentheses. You may omit the empty parameter list for lambda functions without parameters and write: `[] {...}`.

Closure in math set with its boundary (*functional analysis*) or family function, which can be achieved via constructing all possible formulas under basis function (*discrete math*).

But in C++ *lambda closure* is functional object (instance of the class) with defined:
`ret_type operator()(<list of arguments>) const`
The closure type name is not specified by C++, but you automatically deduce if for specific variable via using `auto`. Example:

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

The compiler creates that object during parsing lambda expression. So Lambda's in fact generate C++ classes with `operator()`.

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

You can refer to variables with static storage duration in lambdas. They all 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. `&` - in context of lambda `&` means capture by reference, not dereferencing operator. Example: `[&captureVar1,&captureVar2](int arg1){}`

It exists notation to capture all non-static variables by value, or by reference:

`[=](int arg1){} // capture all not-static vars by value`

`[&](int arg1){} // capture all not-static vars by reference`

Next, a notation exists to capture all non-static vars by value or reference and specify something for other variables. Example of capture all not-static vars by reference, but by value capture Param2:

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

The capture default (`=` or `&`) if used, should always come first. [4, p.752].

Lambda return type can be deduced if 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, and it will be deduced in C++14. A similar trailing syntax can be applied to auto and member - functions.

You can capture only local vars, not member variables of the object. Also, you can allow lambda to modify captured values.

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

You can capture this pointer explicitly

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

or implicitly, because default capture also makes this available.

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

Even though the lambda closure will be an object of a class another class, its function call operator will still have access to all protected and private members of Finder provided with `[this]` pointer to the object.

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.

In C++14, there is an extra feature with a name is **"init capture"**. It allows performing arbitrarily declaration of closure data members:

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

Starting from C++14, you may not want to specify types of arguments. It's called *generic lambdas*. A *generic lambda* is a lambda expression where at least one placeholder type such as `auto`, `auto&`, or `const auto&`. That turns the function call operator of the generated class into a template call operator. Example:

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

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

Starting from C++20, it's possible to be more explicit about the type and define what is called **Lambda Templates**.

Example:

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

Move Semantics

The move brings objects to a valid but unspecified state. Objects, in fact, can be reused after movement.

Move semantics is 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 usually faster. For some objects, there are only move semantics, e.g., thread.

It doesn't make sense to implement a move if (1) its speed is worse or the same as copying, (2) This operation is not on the critical execution path in the program. You shouldn't use `std::swap` to implement the move. It doesn't make any sense.

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

The construction `void f(T&&)` takes precedence over `void f(T&)` if both can be called. C++11 considers `noexcept` calling any destructor, memory allocation. Move self to self - usually not checked because this is not normal behavior. Move operations need not be `noexcept`, but it's preferable.

The First Scary Thing. Handling RValue refs with std::move

The move constructor should be written carefully. The fact is `T&& arg` although is an RValue reference, but in the context of the function body `T&& arg` can be observed as variable, and it makes `arg` as an LValue . The most confusing thing about C++11 due to Scott Meyers.

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

`std::move` - turn 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);
}
```

The Second Scary Thing. Universal references.

You can create a special construction that uses `T&& param` and it is valid only for constructions of the form:

```
template <class T>
void f(T&& arg)
{}
```

That construction is processed by special rules (for details please check Reference Collapsing section of that note):

- 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 (term from Skott Mayers) binds everything:

```
template <class T>
void f(T&& arg)
{}
```

The universal reference is not the official term; Scott Myers coins it only for informal discussions.

The `auto` type deduction working in the same way as template type deduction.

```
int f();
int x;
auto&& z1 = f();  // z1 type is int&&
auto&& z2 = x;    // z2 type is int&
```

In the context of moving two operations are typically needed from C++ Library:

- std::move - unconditional reference RValue cast.
- std::forward - conditional cast of universal references to need reference type. Copies LValue arguments, move RValue arguments. It's applicable only to function templates. Preserves arguments LValueness/RValueness/constness when forwarding them.

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;
};
```

Virtual 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 definition in a derived class must have the same signature as it has in
the base class. The only exception is for the return type that should be the same or as described below. 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].

When you're calling a function using the scope resolution `operator::` that ensures that the virtual mechanism built in the language **is not using**. This call will be resolved at compile time.

When manipulating an object directly (rather than through a pointer or references) and its exact type is known by the compiler, then, in that case, runtime polymorphism is not needed, and the call will be done without using vptr's.

A virtual function invoked from a constructor, or a destructor reflects that the object is partially constructed.

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 change any existing function overrides when the signature of the base class function changes.

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 type of virtual functions.

So it can be a situation when the function is intended to override something, but due to a mistake, it is a new virtual function. But in the case of, using `override` will remove this problem. Better to not repeat virtual in a derived class, but if you want to be explicit, use `override`. B. Stroustrup:
"That it's illogical that virtual is a prefix and override is a suffix. This is part of the price we pay 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&)");}
    virtual void f(){} // declare functions in a base class that can be redefined in
each derived class
    virtual void g(){} // declare functions in a base class that can be redefined in
each derived class
};

struct A: Base
{
    A(){puts(__func__);}
    A(const A&a): Base(a) {puts("A(const A&)");}
    virtual void f(){} // By default, a function that overrides a virtual function
itself becomes virtual
    void g(){}         // By default, a function that overrides a virtual function
itself becomes virtual
};

struct B: Base
{
    B(){puts(__func__);};
    B(const B& b) : Base(b) {puts("B(const B&)");}
   // Better to not repeat virtual in a derived class, but if you want be explicit use
"override"
   // final provides prevent further overriding
   void f() 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.

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.

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 reserved memory of std::vector that encapsulates dynamic linear array.

Documentation: cpp reference details.

3. noexcept function specification

The `noexcept` specification - like `throw()` specification for functions from C++98/03, but it allows more optimization. And 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)`. A special declaration that results in a compilation error if the constant expression expr evaluates to false.

The `static_assert` valid anywhere:

- Global/namespace scope
- Class scope
- Function/block scope.

Example:

```
static_assert(sizeof(void*) == sizeof(int),
              "Pointers and integers are different sizes");
```

Documentation: cpp reference details.

5. alignas operator

The `alignas` operator from (C++11) enforce alignment. `alignas` allocates an object with the requirement to allocate it with alignment suitable for another type. Example:

```
alignas(int) char buff[1024];
```

Documentation: cpp reference details.

6. alignof operator

The `alignof(x)` is the operator built into the language that return the alignment of a type in memory. 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 code and helps us to 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;
}
```

Documentaion: 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:

```
struct S {
[[noreturn]] virtual inline auto f(const unsigned long int* arg)
-> void const {}
};

[[noreturn]] void exit(int);
```

Documentation: cpp reference details.

10. Anonymous Unions

```
  union {
     float f;
     uint32_t d;
  };
  f = 3.14f;
```

Documentation: cpp reference details.

11. Type Alias

Starting from C++11, there are *"Alias Templates"* or *"Smart Typedefs"*.

Using declarations can now be used for "partially bound" templates:

```
// Usual namespace alias from C++98.
namespace AA = std;

// Usual typedef
typedef AA::vector<int> VecInt1;

// Alias for bind some types. From C++11
using VecInt2 = AA::vector<int>;

template <class T>
using MyAllocVec = std::vector<T, MyAllocator>;
// MyAllocVec is alias template
```

Remarks:

- 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. It was not the case in C++03. But since C++11 static variables and local variables are guaranteed to be initialized only once.

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 themselves 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 initializer list of constructor.

If you will add member initialization into the previous code snippet, it will lead to compile error:

```
//...
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 new implicit constructor, that call base class versions. That brings the ability of using `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 of
  // 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 about constructors.

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` is a part of the library and are available from `<cstdint>`.

Documentation: cpp reference about integer types

17. Concurrency Support

For first-time concurrency support has been introduced into the language for C++11. Before C++11 the language was not aware of multithreading aspects.

- std::thread. Independent execution by real OS threads. Thread detachment when no thread joining is needed is supported.
- std::async(). Request asynchronous execution of a function
- std::future<ret_type>. Token representing functions result and encapsulate thrown exception.
- Mutexes (std::mutex, std::timed_mutex, std::recursive_mutex, std::recursive_timed_mutex, std::shared_timed_mutex) for controlled access to shared data.
- Condition Variables (std::condition_variable). To allow threads to communicate about changes to shared data. Via ability to notify one/all waiting threads.

- thread_local data with static storage duration for thread-specific data.
- std::atomic atomic types (e.g., `std::atomic<int>`) with memory ordering options.
- Thread safe only one-time function invocation (std::call_once).
- 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

The `explicit` keyword now applicable to conversion functions. It will prohibit using type conversion implicitly, only explicitly. Example:

```
class MyClass {
  //...
  explicit operator std::string(); // C++11 only
  //...
};
MyClass a;
std::string s = static_cast<std::string>(w)); // ok
```

Documentation: cpp reference about explicit keyword.

19. Current Exception. Internal Details.

The construction in the standard library devoted for for concept of current exception object called *"Exception Pointer"*. Starrting from C++11, it's possible to get it object via: std::current_exception and std::exception_ptr.

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. For example:

```
[[deprecated]] void func(int);
```

Documentaion: cpp reference details about deprecated.

2. Return Type Deduction

For functions, it is now possible to deduce the return type from its return statements.

```
auto f() {
   return 42;
}
```

Documentaion: cpp reference details about auto.

3. Binary Literals

In C/C++, 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;
```

Documentaion: cpp reference details about integer literals.

4. Variable Templates

Example:

```
template<class T>
inline constexpr T pi = T(3.14159)
```

Documentaion: cpp reference details about variable template.

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};
```

Documentaion: cpp reference details about integer literals.

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 about make_unique.

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.

Documentation: cpp reference details structured binding.

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); // from C++17
```

instead of

```
A<int> a(123); // correct syntax in C++98/03/11/14
```

Before C++17, such a feature was supported only for template functions but 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 via using some compile-time expression. This solution offers optimal performance and 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 about compile if.

4. __has_include Macro

The new macro `__has_include` to test that header file is presented. Example of usage:

```
#if __has_include(<optional>)
#endif
```

Documentation: cpp reference details about feature test.

5. std::byte

C++17 introduces byte type to work directly with bytes `std::byte` defined in `<cstddef>`.

Documentation: cpp reference details.

6. fallthrough Attribute

In C/C++ language in `switch` construction, if not explicitly use `break` instruction, instruction flow will fall through after the first `case` statement without reevaluating the predicate. The C++17 added a language feature to signal to the compiler and the person reading your code that you are intentionally using a fallthrough. Example:

```
switch (number)
{
case 1:
 ;
 [[fallthrough]];
case 29:
case 78:
 ;
 break;
//...
}
```

Documentation: cpp reference details about fallthrough.

7. Initialization Statements in if/switch/for

The syntax for define local variable and consequence `if` statement was so common enough that in C++17 have been added a specialized syntax for it.

```
if (auto my=2; my >= 1 || lower <= 2) {
  ;
}
```

The same syntax has been added for `switch` statement:

```
switch (initialization; condition) { ... }
```

The same syntax has been added for `for` statement:

```
for ([initialization;] range_declaration : range_expression)
 // loop statement
```

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 `optional` object to a bool.
- call the `has_value()` member function.
- compare the `optional` object to `std::nullopt`.

Documentation: cpp reference details about 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.
Similar to `std::string_view` there is a `std::span<const T>` that can be used instead of `const`

`std::vector<T>&`.

Documentation: cpp reference details about optional.

10. Inline Variables

Inline variables have been supported only 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 to add static variables into compilable and finally, the linkable binary is under your responsibility.

```
// somewhere in some header file
class ObjectsOld {
    static size_t s_object_count;
};

// somewhere in cpp unit
size_t ObjectsOld::s_object_count;
```

Documentation: cpp reference details about inline keyword.

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

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 subobject 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;
}
```

Documentation: cpp reference details about no_unique_address.

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].

```
operator <=> (const Y&)
```

After defining what is called spaceship operator compiler generates based on that various s (<, >, <=, >=), comparison operators automatically.

The `<=>` operator can return one of the following return types:

- **std::strong_ordering**. It's a enumeration type of available values - std::strong_ordering::less, std::strong_ordering::greater, std::strong_ordering::equal. In mathematical sense it should be used when relation is totally ordered.
- **std::partial_ordering**. It's a enumeration type of available values - std::strong_ordering::less, std::strong_ordering::greater, std::strong_ordering::equivalent, std::strong_ordering::unordered.
- **std::weak_ordering**. It's a enumeration type of available values - std::strong_ordering::less, std::strong_ordering::greater, std::strong_ordering::equivalent.

Documentation: cpp reference details about no_unique_address.

3. likely and unlikely Attribute

The `[[likely]]` and `[[unlikely]]` attributes are applicable for `case` branches in the switch statement to give a compiler a hint to optimize certain branches.

```
switch(value) {
  case 1:
    break;
  [[likely]] case 2:
    break;
}
```

Documentation: cpp reference details about likely.

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);
```

Slightly simplified general form of the format specifiers is the following:

`[[fill]align][sign][#][0][width][.precision][type]`.

Documentation: cpp reference details.

5. source_location::current()

Compile time information about current source file `source_location::current()`. Defined in `<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 about nodiscard.

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. It's not an undefined behaviour.

Documentation: cpp reference arithmetic operations details.

9. Abbreviated Function Templates

Staring from C++20, the template function can be written a bit shorter. That syntax is called *"Abbreviated Function Template"*. 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 so because every occurrence of `auto` in the function parameter list of an abbreviated function template introduces an implicit, unnamed template type parameter. Example:

```
// Classical syntax
template <typename T>
T sqr(T x) { return x * x; }

// Abbreviated Function Templates
auto sqrNew(auto x) { return x * x; }
```

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. Example (from cpp reference):

```
struct A { int x; int y; int z; };
A b{.x = 1, .z = 2};
```

Documentation: cpp reference details

Modules (from C++20)

Standard `#include` preprocessor directive helps organize project but 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 reasons to include it in standards that B.Stroustrup mentioned in his talks is mostly 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.

So the C++20 has introduced that mechanism to form a self-contained subcomponents of related functionality and called it a *module*.

Before technical details, let's take a look into 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** that module.
- A module can **export entire other modules**.
- The combination of all entities that a module export is called the **module interface**.

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 supports it.

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 module interface file
export module simple;

// 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
block.
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.
// Still valid:
// export auto square(const auto& x) { return x * x; }
```

Module Interface File With Separate Implementation

The module interface file then 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);
```

But their definitions, along with any
module-local entities can also 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;
import <string>;
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. [5,p.396]

What you can not Define in a Module Implementation File

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 that import the module quickly.

You import the module via the `import mymodule;` declaration. You do not put angle brackets around the name of your modules, and add you should use a semicolon at the end of the expression, which is not the case when you C preprocessor.

Unlike for 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.

After 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 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.

If you add export in front of an import declaration in module interface files e.g. in the following way:

```
export import <string>;
```

Then any file that imports a module implicitly inherits all import declarations that are exported from that module as well.

Splitting Modules

1. *Simulating Submodules.* C++20 does not formally support the concept of nested submodules, but they can be simulated quite
   easily. The possibility to use 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.
2. *Use Module Partitions.* Submodules can be imported
   individually by the rest of the application because they are just modules. 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. It is not allowed to export any entities

module mymodule:internals;

unsigned int from_c(char c)
{
 // Same switch statement as before...
}
```

To then use the internals 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.

You can partition 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

1. 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 function must have an external link type. Starting from C++20, a template parameter can be of any fundamental type (bool, float, int, and so on), enumeration type, pointer type, and reference type. [4, p.380]. The

compiler needs to be able to evaluate the arguments corresponding to all non-type parameters at compile time.

2. You can use `template` to create 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;
}
```

3. Very rarely, but sometimes while using templates inside templates, there is a need for the `template` qualifier is needed in case of difficulty in understanding.

```
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 at "B.13.6 template" in Special Edition, Biern Stroustrup.

4. An empty `template<>` is syntactically reserved for explicit specialization and cannot be used to create a template without parameters.
5. A template specialization can be complete (type int), or partial (the type that uses T as the basis for something else). However, the general template must be declared before any partial or complete specialization. Also, partial specialization only works for classes but doesn't work for functions. Example:

```
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;
}
```

Template Syntax Remarks

When instantiating a template, starting from C++11 not necessary to make a space in the right shift operator `>>`. In the case of specifying an integral type in a template parameter and wishing 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;
```

Derivation of the template from the list of function arguments
from longer to shorter form A->B->C. 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
template<>
void swap<float>(float* a, float* b)
{
   float temp_temp = *a;
   *a = *b;
   *b = temp_temp;
}
//B
template<>
void swap<>(double* a, double* b)
{
   double temp_temp = *a;
   *a = *b;
   *b = temp_temp;
}
//C
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;
}
```

Default template arguments type must occur at the end for class or variable template. However, for function templates, it's not a requirement because, for functions, there is a rich type deduction template mechanism.

For compilation purposes, you may want to request **explicit template instantiation**. Explicit instantiation of a class template instantiates all members. It's possible to instantiate only individual template class member functions too.

Example of syntax:

```
template class MyVector<int>;
template void f<int>(int&);
template void g(double&);
template void MyVector<int>::size();
```

Also, one more complication - there must be exactly one definition of the corresponding specialization. So if you instantiate a template explicitly in one translation unit, you should not explicitly instantiate it in another.
Example of syntax:

```
extern template class MyVector<int>;
extern template void f<int>(int&);
extern template void g(double&);
extern template void MyVector<int>::size();
```

Let's assume you define template class Array, e.g. in the following way:

```
template <typename T>
class Array {
public:
  explicit Array<T>(size_t size);
  ~Array<T>();
  Array<T>(const Array<T>& array);
private:
  T* m_elements; // Array of type T
  size_t m_size; // Number of array elements
};
```

Using the complete(full) template identifier within a template definition scope is not essential. So the following is the legal code:

```
template <typename T>
class Array {
public:
  explicit Array(size_t size);
  ~Array();
  Array(const Array& array);
private:
  T* m_elements; // Array of type T
  size_t m_size; // Number of array elements
};
```

The same is applied during template class member definitions. After name resolution operator `::` you don't need to repeat the full template class specification. [4, p.638]

```
template <typename T>
Array<T>::Array(size_t size)
: m_elements {new T[size] {}},m_size {size}
{}
```

Variadic Templates

Variadic templates has been introduced in C++11. The motivation is to have the ability to work with a varying number of template parameters. The smallest example:

```
template <class T...>
void f(const T&...arg)
```

Example of performing *printf* in variadic template style (The example is typical and can be found for example in that question in the StackOverflow):

```
template<typename T, typename... Args>
void printf(const char *s, T value, Args... args)
{
    using std::cout;

    while (*s) {
        if (*s == '%') {
            if (*(s + 1) != '%') {
                cout << value;
                s += 2;
                my_printf(s, args...);
                return;
            }
            ++s;
        }
        cout << *s++;
    }
}
```

First thing: In the variadic template for ellipses <`...`> you can put spaces anywhere around it. Construction <`...`> is used for various purposes in the context of variadic templates.

Second thing: The only way to get the next template argument is to recursively call a 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, that construction looks pretty close to that.

Packing operations:

- `class ...Args` - parameters **pack** in template parameters list with optional name. Non-type template parameters can also be organized in the pack via similar syntax `int...`. One more time - whitespace around <`...`> does not matter for the compiler.
- `(Arg&&...params)` - template function arguments parameters **pack**.

Unpacking operations:

- `Args...` - **unpack** parameters. Used inside the body of the template function.
- `sizeof...(Args)` - size of of parameters in terms of number of elements in Args. (It is not the size in bytes).

The last trick with a variadic template is called *fold expression* has been introduced in C++17. The unary syntax for fold expression:

```
template<typename... Args>
bool allLeftFold(Args... args) { return (... && args); }  // Unary right fold (E op
...) becomes (E1 op (... op (EN-1 op EN)))

template<typename... Args>
bool allRightFold(Args... args) { return (args&& ...); }  // Unary left fold (... op E)
becomes (((E1 op E2) op ...) op EN)
```

Documentation cpp reference details

Reference Collapsing Rules and Universal Reference

In C++ you can not declare a reference to reference, but when reference collapsing occurs during template types substitution, there are the following rules:

`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 compiletime
// f<int&>(int&&) => converts into => f<int&>(int&)
f<int&>(i);
```

Example for 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>(Z& && arg) => f(Z& arg)` (see rule (3))

If argument `arg` is RValue reference with type `Z&&` the function will be deduced:

`f<T>(T&& arg) => f<Z>(Z&& && arg) => f(Z&& arg)` (see rule (4))

So in fact, construction

```
void f(Type&& arg);
```

obtains as input true RValue reference.

But construction:

```
template <class T>
void f(T&& arg)
{}
```

can binds everything. The term that has been used to describe that (by Skott Meyers) has obtained a name *universal reference*.

Variants of Casting

`const_cast` - removes `const` and `volatile` modifiers from a pointer.

`static_cast` - casting one type to another using standard or user-defined conversions.

`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 (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 is failed.

`reinterpret_cast` - force casting instructions

`C-style type casting` - tries to use static_cast, if not, uses reinterpret_cast. Further, if necessary, uses `const_cast`.

Private inheritance - in addition to the invisibility of variables by the user of the derived class of members of the vase class, also does not allow the use of implicit type casting.

```
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]:

- Syntactic requirements.

- Semantic requirements.
- Complexity requirements.

The only requirements that we can express with C++ concepts are syntactic ones. Explicitly specify any and all non-obvious semantic and complexity requirements of a concept right now should be done in the form of code comments or other documentation. First, let's describe example how concepts are defined:

```
template <typename T>
concept Small = sizeof(T) <= sizeof(int);
```

General things about concepts:

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* is any expression that the compiler can evaluate at compile time.
4. The constraints in the body of a concept definition are logical expressions connected with `&&` or `||` .
5. Expressions that are connected with logical symbols are **atomic constraint**.

You can define concept in terms of another concepts. Example:

```
template <typename I>
concept Integer = SignedInteger<I> || UnsignedInteger<I>;
```

So concept is defined in the following way:

```
template <parameter list>
concept name = constraints;
```

To express the actual syntactical requirements of a **concept** you can use *requires* expression. In that context *requires* specifies a constant expression on template parameters that evaluate a requirement (since C++20).

```
requires { requirements }
requires (parameter list) { requirements }
```

All the compiler does with requirements is check whether they form valid C++ code or not. The body of a requires expression consists of a sequence of requirements. Each requirement ends with a semicolon and is one of the following types:

- **A simple requirement.**
  Example:

```
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 would compile without errors. All variables you use in these expressions must either be global variables or variables introduced in the parameter list. You cannot declare local variables in the usual way.

- **A compound requirement.**
  Example:

```
template <typename T>concept NoExceptDestructible = requires (T& value) { { value.~T()
} noexcept; };
```

A compound requirement is similar to a simple requirement; only
besides asserting that a given expression must be valid, a compound requirement can also prohibit this expression from ever throwing an exception and/or constraint type that it evaluates to. There is no semicolon after the expression inside the curly braces of a compound requirement. 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 } noexcept -> type-constraint;
```

- **A type requirement and nested requirement.** Example:

```
template <typename S>
concept String = requires
{
 typename S::value_type;
 requires Character<typename S::value_type>;
}
```

Use Concepts

The general outline for a constrained template thus becomes as follows:

```
template <parameter list> requires constraints
template body;
```

Example of using concept to restrict template using:

```
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, declaration keyword **requires** specifying the used constraint.

Starting from C++20 there is a shorthand notation for using concept:

```
template <MyConcept T>
const T& function(const T& a, const T& b)
{ return a > b ? a : b; }
```

Coroutines (C++20)

The execution of function includes the ability to peform the following:

- created stack frame
- execute function
- return value
- delete stack frame

The Coroutines conceptually the same as functions, but they allow two extra actions:

- suspend execution.
- resume execution.

Coroutines in some sense are stackless they reuse stack frame of the caller.

`co_return` - return from coroutine. The `return` statement is prohibited in coroutine. Example:

```
task<int> func(int a, int b)
{
  co_return x;
}
```

`co_await` - suspend coroutine while waiting for another computation to finish.

`co_yeild` - returns a value from coroutine to the caller. Suspend execution of coroutine, until subsequently calling the coroutine.

To execture coroutine until it's not ready:

```
task<int> x = func(1,2)
while (!x.is_ready())
  x.resume();
return x.result();
```

Acknowledgements

Me, 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, and 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 days 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 colleagues and friends from the past with whom I went through various steps of figuring out subtle details of C/C++. Between 2010 and 2020, I have created personal engineering

drafts, some of which are available here (old home page K.Burlachenko). All C/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 that 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 a 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 leaded 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