# C++

Last Updated: March 12, 2023

Primary Source: <https://www.learncpp.com/>

Language Version: C++20, compiled on GCC 12.2.

Major Update: March 2023

Minor Update: None yet.

Additional Notes: Skipped the chapter on Strings and C++20’s features like ranges, spans etc. are still not developed.

1. Terminology
   1. Computer Program

aka application, is a set of instructions to perform some task.

* 1. Source Code

Text written by a programmer to define commands which will be translated into instructions for the computer.

* 1. Execution

When the program’s instructions are executed on the CPU, the stage is called execution of the program.

* 1. Machine Language

Electronic devices (and CPUs) only speak binary (0s and 1s). Then, on top of it, they only understand a few specific instructions and the instructions understandable by a CPU is called an Instruction Set. Not all CPUs support the same IS’.

For ex.:

00100100 00011111

* 1. Assembly Language

A bit more understandable than ML but requires a lot of instructions, plus it is not guaranteed 2 machines will support the same assembly instructions. Assembly Code is translated into ML using an assembler.

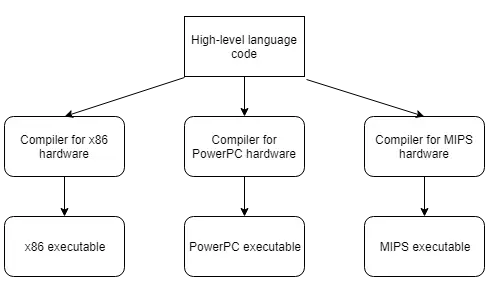
For ex.:

mov al, 061h

* 1. High Level Language

Raw English instructions, these are very straightforward to understand. Languages like C++, Python, Perl etc. are all HLLs.

The benefits to HLLs are that the source code doesn’t need to bother about the supported CPU instructions at all, and that is the job of its translator. And that also means the HLL itself can be abstractful and a single instruction can map to a lot more instructions.



However, even so many HLL instructions can still be platform-agnostic like a C++ header could be only available on Windows OS’ and so on. In that case, the source code becomes platform dependent.

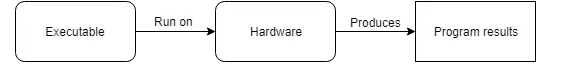
One interesting thing to note here is that the HLL’s rules for isntructions are defined by the translator. So if a translator supports only a certain HLL instructions then it won’t translate other translator’s instructions so a single language can have multiple ‘rulesets’ so to speak. In C++’s case, we have MSVC compiler and mingw as 2 famous compilers and we can see that not all C++ Instructions written to be compiled on MSVC work on mingw and vice versa.

Mingw is based on GNU C++ compiler, also called g++ (which calls gcc.exe when compiling C programs)

* + 1. They also need to be translated into ML, it is done using mainly 2 ways
       1. Compiling

Source code is translated into ML instructions using a compiler, which also optimizes the code to be much faster and efficient than it’d be when manually done.



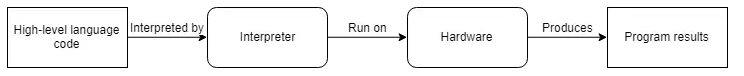


Compiling is a one-off task, it doesn’t need to be done again for already compiled code. But compilation itself is a relatively (relative to interpreting) slow process generally.

C++ is a compiled lang.

* + - 1. Interpreting

Source code is directly executed as each instruction that is being executed is only translated into ML instruction. This makes the execution slower than compiled instructions but there is no time spent on compilation, however the code needs to be re-interpreted each time it is executed.



JS and Python are good examples of interpreted langs.

There are also langs that mix the 2 methods like Java.

* 1. Project

A project is basically a container to contain all the source code files, images, config files, headers etc.

A single project means a single program in simpler terms, and also as a good architecture principle, a project must be a single program. That is a project that aims to process audio files mustn’t try to also run a web server.

* 1. Solution or Workspace

A single solution or workspace may contain multiple projects, this is to segregate different parts of an idea into different projects and not have single containers for all of them which may also interject into each other’s functionalities.

For ex.:

We can have a project that processes audio files, and another that runs a web server to serve those files. But say we have yet another web server, then if both the web servers exist in the same project, it may be possible that the ports used by one are also in use by the other causing them to malfunction.

* 1. The Standard

C++’s version specifications are laid out in a ‘standard document’ that is a paid resource. But before the final version of the document is made, drafts are proposed and merged, the final draft is close enough to the final version or the standard document.

Both can be found here

<https://isocpp.org/std/the-standard>

* 1. Expression

It’s a combination of literals, variables, operators, etc.

For ex.:

int c{ (2 \* 3) + 4 }; // initialize variable c with computed value 10

This line is a statement and the computation inside is the expression.

The process of executing an expression is called an evaluation and it yields a result, aka the expression’s result.

For ex.

x=5

Here the expression evaluates and only the variable is left, finally the result of the expression is the value of the variable.

Similarly,

2 // 2 is a literal that evaluates to value 2

"Hello world!" // "Hello world!" is a literal that evaluates to text "Hello world!"

x // x is a variable that evaluates to the value of x

2 + 3 // operator+ uses operands 2 and 3 to evaluates to value 5

five() // evaluates to the return value of function five()

An expression doesn’t end with a ;

So

x=5

is ok, it uses the = operator to copy 5 to x and then the expression here yields x.

* 1. Execution Path

The sequence of statements that a CPU executes.

* 1. Unit Testing

Testing individual units of a program, like functions.

* 1. Integration Testing

Testing combined units of a program, such as testing ‘download’ feature of an app.

* 1. Code Coverage

Describes how many units(in %) of the source code were tested or have tests defined for them.

* 1. Statement Coverage

Describes how much (in %) of the statements in the source code are covered by tests.

* 1. Branch Coverage

Describes how many (in %) of the branches in the source code are covered by tests.

* 1. Loop Coverage

..same for loops..

* 1. Precondition, invariant and postcondition

Precondition is a condition that checks if an statement would be successfully executed before the execution itself.

For ex.:

int y{2};

if (y>2) //precondition

{

std::cerr<<”cant do”;

return;

}

y/=2;

Postcondition

Similarly, here we simply check if the execution was successful.

Invariant

Condition that’s true whilst the execution is going on, for ex. checking value in an array.

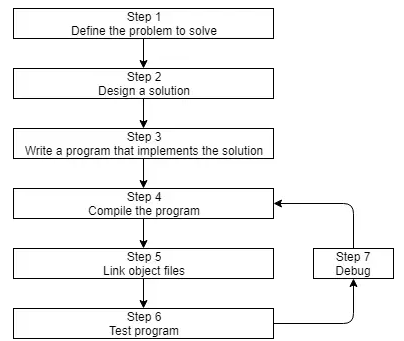
1. About C++

Successor to C, which was built by Dennis Ritchie in 1972 at Bell Labs. C++ was build by Bjarne Stroustrup in the same lab in 1979 following the major success of C as an extension. It was standardized by ISO committee in 1998, i.e., ISO-C defined the rules all C++ compilers would have to follow. Ever since 2011, each 3 yrs an updated specification for C++ is released called its ‘language specification’.

Currently we have C++11, C++14, C++17, C++20 and works are underway for C++23.

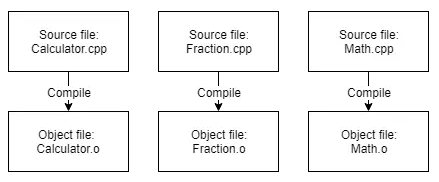
C++’s design philosophy can be summed up with “Trust the Programmer”.

1. C++ Program’s development lifecycle



* 1. Step 1 to 3 are same for every language. Step 6 and 7 are too, 4 and 5 are where each language is a bit different.
  2. Compilation

Does 2 jobs, first checks if the source code is following all its rules. If there are no errors, it translates the code into Object (.o) files.



* + 1. Problems during compilation

If the source code fails to obey the language specification (or compiler rules) then the compiler will throw an error. This halts the compilation.

And if the compiler expects a certain rule to be followed but it can work with the given source code then it may present a warning. Warnings don’t halt compilation.

For ex.:

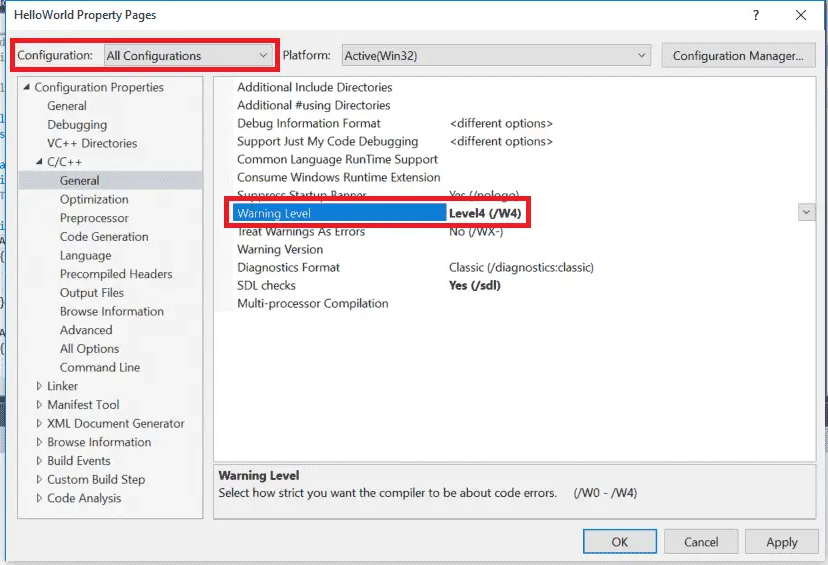
int a=2;

size\_t b=a;

is a warning because size\_t can only take +ve ints while the int can take both +ve and -ve ints.

We can either fix the warning-causing line or disable the specific warning. The way to do this is not standardized and varies between compilers.

Warnings are categorized according to levels, the higher the less serious the warning is, and if we configure to display a higher level of warnings, the more verbose compiler is about warnings. We can configure what level of warnings to display using -W flag or in Visual studio using this option



Note: All is an unrecommended level.

* + 1. Assembly

Generally source code is directly compiled into machine code in the form of object files, but it is also possible to generate assembly code. These files have .s extension.

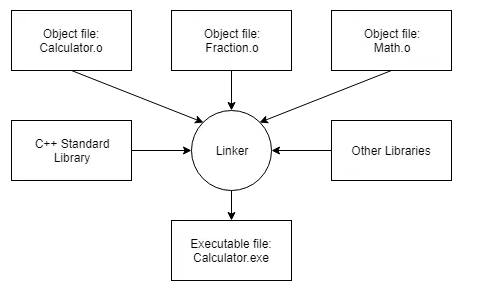
To get assembly file,

g++ -S -o file.s file.cpp

generates assembly file.

* 1. Linking

After object files are created, the linker kicks in and links all the object files, then it links all the library files which is a collection of precompiled code and finally it resolves all dependencies so that each file that requires another file properly links with the file. Finally the output is an executable that can directly be executed on the host machine.



Custom Compilation and Linking steps can be given in C++ using a makefile.

* 1. The entire process of preprocessing, compiling and linking is called translation. And there are various phases (9 as of now) of translation called translation phases.

Given here: <https://en.cppreference.com/w/cpp/language/translation_phases>

1. Sample Program

#include <iostream>

int main()

{

std::cout << "Hewwwo Wowrld :3";

return 0;

}

It is saved as a .cpp file. A single project can have multiple .cpps but 1 is necessary.

If it is saved in a main.cpp file then it can be compiled with

g++ -std=c++17 -O2 -Wall main.cpp -o main

and then ran with simply

./main.exe

1. Configuration

The solution, project, compiler, linker etc. support various options.

* 1. Build Configuration

aka Build Target. It’s the configuration for the compiler. By default, IDEs generate 2 BTs, release configuration and debug configuration, release config is intended to optimize execution speed and smaller executable size at the cost of compilation time.

BT can be defined as args to the compiler, for example

g++ -std=c++11 -O2 -Wall test.cpp -o test

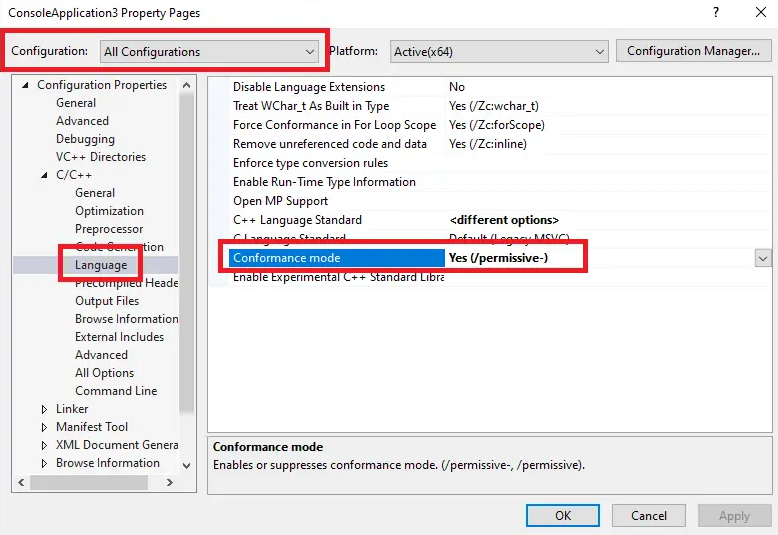
Here we pass the -std -o and -W to G++ as BT for test.cpp.

* 1. Compiler-Specific Configurations

Compilers also have their own specific behaviors, and using a specific compiler’s behavior might require the source code to be incompatible with other compilers. These compiler specific behaviors are called compiler extensions and are provided by compilers to give extra functionality.

It is recommended to disable them to ensure the source code doesn’t rely on compiler-specific behaviors and remains true to the language standard.

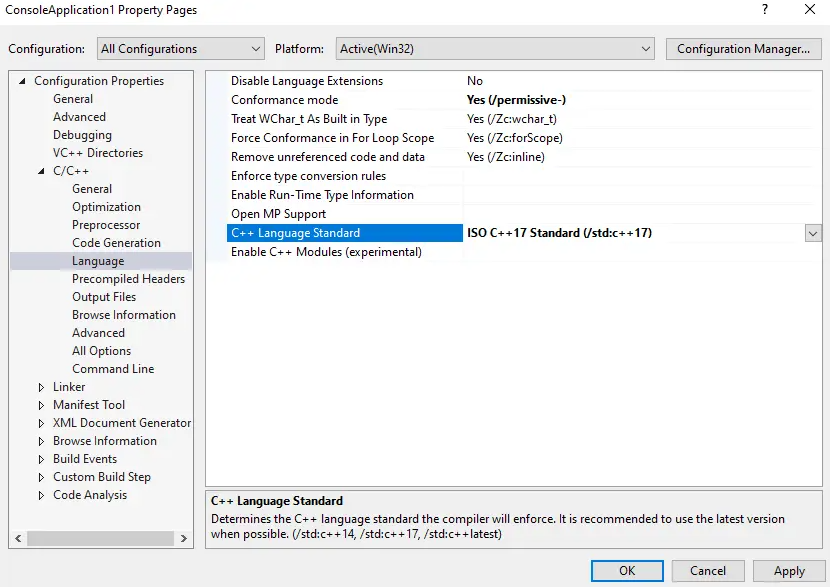
In Visual Studio, these are defined as “Conformance Mode” and are by-default on.



* 1. Language Version

Since C++ has various versions, we can specify which one to compile against.

In G++, we use the -std flag, for example -std=c++20. In Visual Studio



* 1. To include multiple locations where g++ searches for header files

g++ -I ../Helpers/Easybench -o points PIP.cpp ../Helpers/Easybench/Easybench.cpp

Here, -I includes Easybench folder in its known dirs. when it has to look for a header file.

Then PIP.cpp and Easybench.cpp are compiled and whatever header they use will be searched for in the default dirs. and the given dirs..

* 1. Debug Symbols

To include debug symbols in the generated executable,

g++ -g -o points PIP.cpp

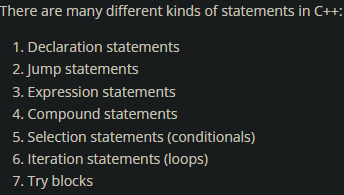
-g does that.

Normally, when we generate a binary executable, it is simply a host of mangled instructions like int i=2; would be translated to set memory address 0xsomething to 2. Which is not human readable and doesn’t really know about ‘i' after compilation. A symbol can be a variable name, function name etc. and whilst the binary does have symbols, these are mangled and highly optimized.

A debug symbol is a symbol that contains information about the name as in the source file as well and using a symbol mapping table (which can be external to the binary or inside it), it maps which symbols in the binary correspond to which names in the source file. This helps when in debugging we wish to know what a variable or symbol goes through.

1. Statement

Each instruction for C++ is called a statement in C++, generally they end with semicolon. But there are different types of statements so not always do they end with semicolons.



1. Preprocessor Directive

PPDs are special statements that are processed in the preprocessing stage, which occurs before the compilation stage. These directives do text manipulation and affect the file before it is actually compiled.

They start with # and end with a newline

For ex.

#include<iostream>

This means that the iostream library must be opened up and linked with this file.

PPD don’t follow C++ syntax rules and have their own rules and can be anywhere in the code.

* 1. Preprocessors

Before the actual source code is compiled, all the preprocessors are processed, this stage is called the preprocessing and does a few things such as removing comments, checking proper line termination, etc. All of which takes place in a temporary or in-memory file called a translation unit and not the original source code. The compiler then compiles the TU.

* 1. #include

This PPD takes a filename and then replaces the PPD with the actual contents of the given file.

The TU has all the contents from the source file + all the contents of all the #include files.

For ex.

#include<iostream>

* 1. #define

This PPD defines a macro. This is a simple rule that replaces a given text with a provided text everywhere in the source file.

There are 2 kinds of macros in C++,

Object macros, to define objects

Function macros, to define functions

Object macros look like so

#define int AA

and now we can use

AA a;

and after preprocessing the TU will have

int a;

#define ABC

is also ok, it replaces the occurrence ABC with nothing in the TU.

* 1. Conditional PPDs

There are 3 common PPDs that are used to do something based on a condition, #ifdef, #ifnded and #endif

For ex.

#define X

int main()

{

#ifdef X

std::cout<<”yo”;

#endif

}

ifdef and endif blocks tell preprocessor to include the lines in between only if the identifier is defined. So here cout will be available in the TU and hence will be compiled. But if X wasn’t defined then the line would be removed and cout won’t be compiled. By not compiled I mean the lines would be removed, and the rest of the code will remain the same.

ifndef is the opposite of ifdef. So if X wasn’t defined then it’ll include the given lines else not.

There’s also

#if 0

std::cout<<”yo”;

#endif

This won’t include the lines within.

#define doesn’t cause text substitution for other PPDs. So,

#define FOO 9

#ifdef FOO

std::cout << FOO << '\n'; // This FOO gets replaced with 9

#endif

Here #ifdef FOO doesn’t have its FOO replaced, but the one in the normal code does.

* 1. PPDs don’t care about scopes or positions

For ex.:

void pp(){

#define X 9

}

int main() {

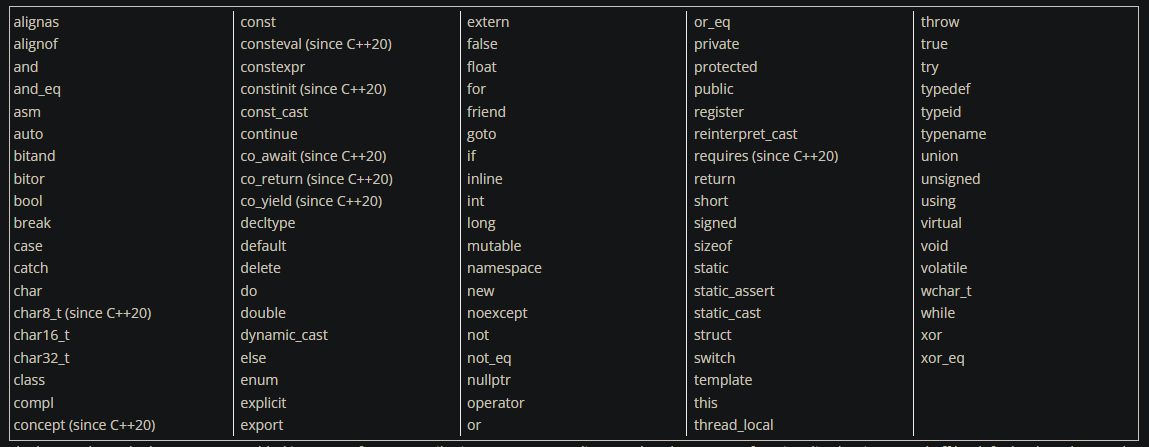
std::cout<<X;

}

works, as PPDs are resolved top-to-bottom in the preprocessing stage and don’t care about scopes. In the TU, the X is already replaced by 9.

However, just like compilation, PPDs are ‘forgotten’ after a source file has finished preprocessing. They don’t affect other source files.

1. Keywords



are the reserved keywords in C++.

A reserved keyword is called so because wherever the word is used it defines a special behavior, this is why keywords can’t be used for variable/function/type/etc. names.

There are some other special words too such as

override, final, import and module.

These are not reserved as they only have special meaning in certain contexts.

Keywords are special because we don’t need any headers or libraries to use them, they can be used directly and compiler understands their purpose right away.

1. Whitespace

In C++, whitespace is ignored except when inside “quotes”.

For ex.:

 std::cout << "Hello world!";

std::cout << "Hello world!";

std::cout << "Hello world!";

std::cout

<< "Hello world!";

std::cout

<< "Hello ”

“world!";

are all the same thing.

but

std::cout<<”Hello

world”;

is an error.

1. Declaration vs. Definition

Declaration and Definition are different things in C++. A declaration is a statement that tells the compiler about the existence of an identifier and its type information.

For ex.:

int x;

This is a declaration of a variable x of type int. This is also a definition in this case.

A definition is an implementation of an identifier.

For ex.:

void add() {

…

}

is a definition of function add.

Definitions satisfy the linker, as it links the definition with the identifier.

Declarations may not be definitions but definitions are always declarations.

For ex.:

int add();

is a forward declaration but not a definition. Such identifiers are called pure declarations.

* 1. One Definition Rule

There can be as many declarations but there must always be a single definition.

The ODR is a rule to make sure ambiguous definitions aren’t present, and there are 3 levels to ODR.

1. In a given file, variables, types, functions and templates must always have single defn.
2. In a given program, variables and functions must have single defn.
3. Types, templates, inline variables and functions are allowed to have multiple defns.

Whilst 1 and 2 are outright errors, 3 isn’t an error but may lead to undefined behavior.

* 1. Declarations

Simply declaring that an identifier exists.

For ex.:

int x;

is a declaration.

1. Comments

// for single line

/\* multi-line

\*/

1. Objects

In HLLs, data is stored in objects, which are abstractions over the actual memory addresses used to store the data.

For ex.

Instead of saying “store 4 in address xyz”, which is error prone, we say “store 4 in object x” and the actual address in the memory where it is stored and other details are abstracted away from the programmer.

In C++, this object is called a variable.

For ex.:

int x=2;

Here x is an object of type int and value 2, and at runtime (the time of execution of code), the value 2 is stored in a memory address that is managed automatically.

* 1. Variables

In C++, variables are declared like so

<data type> <variable name>;

so like

int x;

Multiple variables can be declared at the same time like so

int x,y;

They can be assigned values like so

x=2;

* 1. Initialization

We can also define values at the time of declaration

int x = 2; //copy initialization

int x{2}; //direct list initialization

int x(2); //direct initialization

int x={2}; //copy list initialization

int x{}; //value initialization

That is, there are 5 ways to initialize variables at the time of declaration. They all behave slightly differently but it is recommended to use direct list initialization if we wish to initialize with some value or use value initialization, in which case a default value of the type is assigned to the variable (0 for int).

It is unrecommended to use default initialization,

int x;

as this leaves an indeterminate value in the variable, also called garbage value. They aren’t null, infact C++ has no concept of null. A variable will always hold a value of its type.

NULL itself is defined as 0 or void \* and is a separate type altogether.

When a variable is declared, it is assigned a memory address, and if we don’t pass a value (or default value), the variable has that address’ value in it, which is why its called a garbage value.

The reason this behavior still exists is because if there are thousands of variable created by a program then initializing them all with a default value adds additional time complexity and it can be avoided if we leave the garbage values until we assign them some value.

Similarly, we avoid copy initialization because for complex types ‘copying’ may not be the most efficient thing. The value on the right is copied to the left, with the assignment operator ‘=’.

For multiple variables we can do this way

int a = 5, b = 6; // copy initialization

int c( 7 ), d( 8 ); // direct initialization

int e { 9 }, f { 10 }; // direct brace initialization (preferred)

int g = { 9 }, h = { 10 }; // copy brace initialization

int i {}, j {}; // value initialization

Something like

int a, b=10;

might be an error in C++.

But not

int a, b( 5 );

int c, d{ 5 };

* 1. Unused Variables

Generally it is advised to not have unused variables in the code as they add to the complexity of the code for no reason.

And they may be treated as warnings/errors,

Using this attribute that can be avoided

[[maybe\_unused]] int x { 5 };

1. I/O Console

Input and Output to the Console is handled using the Standard Library.

Primarily, we have 2 ways of I/O for console, C++’s cin and cout, and C’s scanf and printf.

* 1. Streams

In C++, I/O is implemented using streams. A stream is basically a sequence of bytes. The source could be things like keyboard presses or timers and the sink (terminal point) could be our program.

The input stream, handled by istream class from the ios class (with ios\_base as its base) is how we read data entered by the keyboard at runtime. We use the extraction operator >> to extract data from it.

The output stream is handled by the ostream class. The insertion operator << inserts data into this stream.

The iostream class provides both these classes and is primarily used for I/O in C++.

Standard Stream

It’s a pre-connected (source/sink connected) stream provided by the environment itself. C++ has 4 of these,

cin: istream, Tied to the standard input (like keyboard)

cout: ostream, Tied to the standard output (like console/terminal)

cerr: ostream, Tied to the standard error (also console), it is unbuffered.

clog: Like cerr but is buffered. Rarely used.

Buffered means it is first written to memory then the OS decides when to clear a segment (a chunk of data). Unbuffered skips the memory and interacts straight with the program.

* 1. IOStream

We can use

#include <iostream>

to be able to use cin and cout.

For ex.:

std::cout << "Hello world!"; // print Hello world! to console

cout can print numbers as well.

and we can concatenate variables and strings in cout with multiple << operators.

cout means character output.

For ex.:

int a{2};

std::cout << 4 << a <<”yo” <<std::endl;

endl inserts a new line and flushes the buffer.

In the case of cin and cout, they use IO buffers, this is like a list of things to be written to the console and is periodically read by the system. This means our cout doesn’t directly write to the console, instead it passes it to the buffer which then writes to the console.

When the entire buffer is written to the console, the buffer is reset, this is called flushing the buffer.

std::endl is a bit slower than ‘\n’, if we want to insert a newline we can use ‘\n’ instead. The reason it is slower is because it flushes the buffer each time it is executed which disrupts the buffer’s normal execution.

Similarly, we have cin,

int x;

cin>> x;

cin waits for an input to the console and then passes it to the provided variable.

and similar to cout we can use multiple variables with >> to read multiple values from a line.

 cin>>x>>y;

* 1. cin

When we indicate an extraction operator “>>” in our code, the program expects an input from an input stream, cin is an input stream to take input from the console or rather, from the buffer. Then, when we enter text in the console it is put in a buffer, which is a region of memory allocated to temporarily hold the input data until it is passed back and then it is cleared. Finally the buffer is cleared and its values “extracted” into the input stream.

If there’s data already in the buffer then it is directly passed to the input stream without asking the console for input.

cin is a stream that only extracts data that the target variable holds and ignores whitespace, newlines, tabs, spaces generally, so

for ex.:

int x{};

std::cin>>x;

and we enter 2a then press enter

then 2 is stored in x and “a\n” is left in the buffer.

Similarly,

#include<iostream>

int main() {

char c;

while(std::cin>>c){ //reads all characters from the input stream

std::cout<<c;

}

}

If the input is “Hewwo Wowld”, it prints “HewwoWowld” because cin ignores whitespaces,etc. The reason std::cin>>c returns True for the while loop to run is because of how the extraction operator >> works for std::cin, it returns c and as long as there is any character in c, it will be true.

* + 1. streamsize and ignore

Each char in the input stream takes std::streamsize size, the max. no. of chars that can be stored in an input stream can be retrieved with

std::numeric\_limits<std::streamsize>::max()

So, if we have some value in buffer and we want to ignore all values in it we can use

#include<limits>  
std::cin.ignore(std::numeric\_limits<std::streamsize>::max(),’\n’);

numeric\_limits simply contains min and max values that can be stored by a type.

cin.ignore discards given no. of characters in the buffer unless it finds the given character first. So here, it ignores all chars in the buffer unless it finds ‘\n’ first.

* + 1. Fail mode

If extraction fails then cin goes into failed mode.

For ex.:

int16\_t x{};

cin>>x;

and if we enter 100000 then cin will check but find out the value is too big for the type, so it will assign the nearest limit value, which is 32767 for int16\_t and go into failed mode. If it can’t even extract the value into the type it will zero-initialize the variable.

Nothing really changes except for a failed flag to be set in cin. We can check if cin is in failed mode with

if(!std::cin) //returns failed status, cin is false if failed, true otherwise

{…}

std::cin.clear(); clears failed status and sets cin to be in normal mode again.

* + 1. cin manipulators

A manipulator is an object that, when applied to a stream using the extraction or insertion operators modify its behavior.

std::endl

This manip prints a newline and flushes the buffered output.

std::setw(<int>)

Defined in #include<iomanip>, applies a limit to the max no. of characters read from the input stream.

For ex.:

#include<iomanip>

#include<iostream>

int main() {

char c[10];

std::cin>>std::setw(10)>>c;

}

Here, even if input has 20 characters, it will only read the first 10.

cin.get()

This method of cin doesn’t ignore whitespaces, newlines, tabs or spaces.

For ex.:

#include<iostream>

int main() {

char c;

while(std::cin.get(c)){

std::cout<<c;

}

}

For “Yo a”, it prints “Yo a” back.

It has an overload that accepts a character array or string too,

#include<iostream>

int main() {

char c[10];

std::cin.get(c, 10); //Remember, the 10th char here will be a newline character

std::cout<<c;

}

works the same way. Only C-Style Strings have to worry about newline character at the end, individual characters don’t have to worry.

A key thing to remember about cin.get() for a C-style string is that it reads the given no. of chars but stops early if it finds a newline and then leaves the newline in the input stream, meaning if we call a cin.get() after another without clearing the input stream in between we will have an empty input.

For ex.:

#include<iostream>

int main() {

char c[10];

std::cin.get(c, 10);

std::cout<<c;

std::cin.get(c, 10);

std::cout<<c;

}

It will only ask for input once, so if we enter “Yo”, then a newline character will be left in the input stream and 2nd cin.get() will stop even before reading anything. So the output here is “Yo” and “”.

cin.getline()

The same as cin.get()’s C-Style string overload, however it even reads in newline so it is not left in the input stream.

On a related note, there is a special version of cin.getline() just called getline defined in #include<string> specifically for reading in std::string objects.

For ex.:

#include<iostream>

#include<string>

int main() {

std::string s;

std::getline(std::cin, s); //takes input stream as argument

std::cout<<c;

}

Works just like cin.getline() but is meant for std::string.

cin.ignore(<optional int>)

Discards the given no. of chars from the input stream, if int not given then just discards the first one.

cin.peek()

Reads a char from istream without clearing it from the istream.

cin.unget()

Sends the last read character back into the start of the istream allowing it to be read again.

cin.putback(<char>)

Puts the given char in the istream’s front.

Other manips here: <https://en.cppreference.com/w/cpp/io/basic_istream>

* 1. cout flags and manipulators

For cout we can use manips as well as another thing called flags, these are like boolean switches that can be turned on or off for the cout.

To set/enable a flag, we use cout.setf(<flag>)

We can enable multiple flags at a time by ORring them, like cout.setf(flag 1 | flag 2)

To unset/disable a flag, we use cout.unsetf(<flag>)

Can also take multiple flags using the Bitwise OR.

Format Group

Some flags belong to groups, these groups perform similarly so they have their flags grouped together. For ex., the basefield format group contains the flags oct, dec and hex, which define what the base of output numbers will be, but if we have dec on (default) and also turn hex on then this won’t do anything because they both do the same thing so we need to turn dec off before turning hex on to make cout have base hex. Alternatively, cout.setf() also accepts an overload with a format group, so then it turns this flag on in its format group and all others off.

Flags:

These live in the std::ios class.

std::ios::showpos flag

Outputs + in front of +ve numbers.

For ex.:

#include<iostream>

int main() {

std::cout.setf(std::ios::showpos);

std::cout<<27;

}

prints “+27”.

std::ios::hex, std::ios::dec and std::ios::oct

Defines the output number’s base.

For ex.:

#include<iostream>

int main() {

std::cout.setf(std::ios::hex, std::ios::basefield); //the latter param is the format group

std::cout<<27;

}

prints “1b” which is 27 in hexadecimal.

This is how format groups are defined and used.

For cout, we can also use manipulators,some of which are the same name as flags, the benefit is that we don’t need to specify format groups or set and unset, we just define a manip and it automatically unsets all the other in its group. However some flags are only available to setf whilst some are only available to manipulators.

These live in plain std namespace.

For ex.:

#include<iostream>

int main() {

std::cout<<std::hex; //notice the lack of ios

std::cout<<27;

}

prints “1b” as expected.

There are a whole bunch of ostream manipulators and flags:

<https://www.learncpp.com/cpp-tutorial/output-with-ostream-and-ios/>

* 1. Streams and Strings

Unlike cin, cout and the like streams for strings simply provide stream like behavior over strings without connecting to any external sources.

We have 6 primary streams for strings, called string streams, all defined in #include<sstream>

istringstream (derived from istream)

ostringstring (from ostream)

stringstream (from iostream)

wistringstream: just like istringstream but for wide characters

wostringstream

wstringstream

For ex.:

#include<iostream>

#include<sstream>

#include<string>

int main() {

std::stringstream s{};

s<<”yoo \n”; //insert this string into s, this appends “yoo \n” to s’ buffer.

s.str(“yalo”); //replaces the string in buffer with “yalo”

std::cout<< s.str(); //prints “yalo”, .str() reads the buffer

s.str(“yoo halo”);

std::string sr;

std::string sr2;

s>>sr; //puts “yoo” in sr

s>>sr2; //puts “halo” in sr2

std::cout<<sr<<” “<<sr2; //prints “yoo halo”

}

The difference between getting a string out of std::stringstream using .str() vs. extraction op>> is that .str() gets the whole buffer whilst extraction op>> gets the next extractable string ignoring newline, tabs, spaces and whitespaces just like we see cin. And just like we see in cin, the extraction op>> only gets the value that can be extracted and applied to the variable from the stream.

Clearing std::stringstream’s buffer

.str(“”) or .str(std::string{}) and then calling .clear();

Just like cin has error flags set in it, so does std::stringstream, .clear() resets these flags.

* 1. Stream States

The stream states for all streams, be it string stream or cin, are given by flags in the std::ios class.

These are the states

goodbit: All OK

badbit: Fatal error, like reading past the end of stream

eofbit: Stream at end of file

failbit: Non-Fatal error, like user input is double whilst int was expected.

And to know if a state is true, we call these methods on the stream object

.good(): True for goodbit

.bad()

.eof()

.fail()

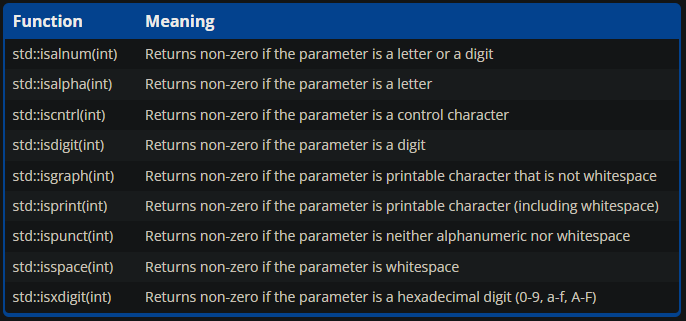
.rdstate(): Returns currently set flags

.setstate(<state>): Sets this state to true

We can use .clear() to reset the stream state. Or .clear(<state>) to reset the stream state and set this state to true.

* 1. Input validation

Defined in #include<cctype>, C++ provides us methods to check if a given character is a given type. This is faster than regex (which is available in C++ too).



* 1. File I/O

Uses streams and works similar to other stream handlers like cin to read and/or write to files.

Defined in #include<fstream>

We have

ifstream (derived from istream)

ofstream (from ostream)

fstream (from iostream)

Some care needs to be taken for fstream objects as opposed to cin, such as setting up fstream objects, reading/writing and setting parameters and then reading/writing to files and finally closing the fstream object.

From C++17, file I/O is better done using std::filesystem.

For ex.:

#include<fstream>

#include<iostream>

int main() {

std::ofstream f1{“test.txt”};

if(!f1) { //if goodbit is false

std::cout<<”Failed to open test.txt for writing”<<’\n’;

return;

}

f1 << “yoo \n”;

}

works. We didn’t need to call f1.close() explicitly as it is called through it’s destructor anyway. ostream can open a file for writing even if doesn’t exist. Then it creates the file.

For input

#include<fstream>

#include<iostream>

#include<string>

int main() {

std::ifstream f1{“test.txt”};

if(!f1) { //if goodbit is false

std::cout<<”Failed to open test.txt for writing”<<’\n’;

return;

}

std::string s{};

while(f1) { //while goodbit is set, meaning a character was read successfully

std::getline(f1, s); //gets each line (separated by \n) into s

std::cout<<s<<’\n’;

}

}

Buffers and crashes

When the program terminates or crashes early, the destructors of the stream classes are not called. This means the streams aren’t closed. Another thing is buffers, normally the output is buffered, meaning it is on the memory and not written right away. We can call .flush or std::flush() or std::endl on an output stream to flush the buffer explicitly, calling .close() also calls the .flush().

So, it is recommended to call .close() on file streams because, firstly they ensure the file is closed at the end of the program and secondly flush the output too.

open()

Just like ofstream, ifstream and fstream can .close() a stream, they can .open() another explicitly as well (but only after one is closed). This method accepts the same params as their ctor.

Similarly, .is\_open() returns true if a file is open.

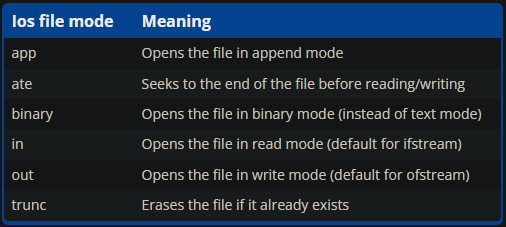
.remove()

Deletes a file.

* + 1. File Modes

These define how a file is read/written to. These are passed as param to the fstream, ifstream and ofstream ctor.

These are the file modes



Yes they are defined in the std::ios class.

We can OR them to enable multiple at once.

The fstream ctor that just accepts a string (file path) by default does std::ios::in | std::ios::out.

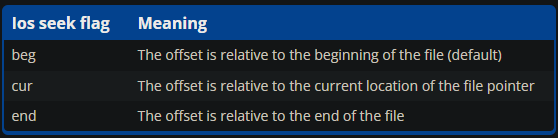
Due to the way fstream is implemented, it will fail to open a file for write if it doesn’t exist since it also have std::ios::in set.

* + 1. seekg() and seekp()

.seekg() is applicable to ifstream objects and .seekp() is applicable to ofstream objects, tese methods seek the read/write pointer to a given byte in the file byte sequence (stream).

These methods also take a seek flag which determines the relative byte position.

These are



Yes these are defined in the std::ios class.

For ex.:

#include<fstream>

#include<iostream>

int main() {

std::ofstream f1{“test.txt”, std::ios::out};

if(!f1) {

std::cout<<”Failed to open test.txt for writing”<<’\n’;

return;

}

f1 << “yoo \n”;

f1.seekp(2,std::ios::end); //puts the file pointer 2 bytes after the end byte, this is an invalid position

f1.seekp(-2, std::ios::end); //2 bytes before end byte

f1.seekp(3); //beg by default, 3 bytes after start

}

There’s also .tellg() which returns the current byte position from the beg, it’s used with fstream objects. fstream objects can both read and write but they can only do one thing at a time, to switch to the other mode a seek operation must be done (odd behavior).

For ex.:

#include<fstream>

#include<iostream>

int main() {

std::fstream f1{“test.txt” };

if(!f1) {

std::cout<<”Failed to open test.txt for writing”<<’\n’;

return;

}

f1 << “yoo \n”;

f1.seekg(f1.tellg(), std::ios::beg); //now we can do istream operations on fstream f1

}

1. Constant

A value that never changes during runtime.

* 1. const

This keyword turns a type into constant, meaning its value cannot be changed after initialization.

For ex.:

const int yo{2};

int const alsoyo{3};

Both are valid in C++.

It can also be used in function parameters

void add(const int yo)

{

…

}

add(2); //works

However this should be avoided unless necessary for normal types as this requires additional compiler resources to ensure const is followed inside the function.

Similarly const can also be applied to return type of functions

const int add(){

return 2;

}

This should also be avoided as it may impede some compiler optimizations.

* 1. Symbolic Constants

These are the constants that are defined using symbols, such as using #define PPD.

#define YO 2

is ok, but the problem with Object macros is that it can conflict with other parts of the program and replace any of its occurrence, so if a lot of trivial Object macros exist, then it is highly likely that some identifier in our code that shouldn’t be replaced will be.

Like if we have a

int YO {4};

then the preprocessor will replace YO with 2 and result in compilation error. The likelihood of such an error only increases with the no. of files that include our file in their project.

This is why it is recommended to use other constant types than these.

* 1. Compile-Time constants and constant expressions

CTC are simple values known at compile time, such as

const int x{2};

CTEs are expressions that can be evaluated at compile time, such as

int x {2+3};

may be optimized at compile time and resolved as 5, otherwise each runtime will require the processing of this expression.

To make sure this expression is always evaluated at compile time we define it as constant

const int x{2+3};

will now always be evaluated at compile time.

These CTC formed from expressions should be used as much as possible and even used to replace symbolic constants as the compiler can optimize this and replace occurences of x with the value everywhere making sure the variable doesn’t even consume any memory.

In subexpressions, such as

int y{};

std::cin>>y;

std::cout<<y<<” yo ”<<2+3;

we know “ yo “ and 2+3 are CTCs but y isn’t, in these cases too the compiler evaluates the CTC subexpressions at compile time. This process is called Constant Folding.

* 1. Runtime Constant

CTC are known at compile time, but we may also have constants that are only known at runtime, these are known as RTC. These are also constants as they can’t be modified once initialized.

For ex.:

int pp(){

int x;

cin>>x;

return x;

}

const int d {pp()};

d is a constant but its value isn’t known until runtime.

* 1. Constexpr

RTC and CTC are both constants but CTC are the ones that are actually optimized at compile time. Still, the ambiguity of both of their syntaxes makes it hard to know which is being used. This is why we use constexpr keyword, to indicate to the compiler that this is a CTC and must be evaluated at compile time, if it can’t be evaluated at compile time then this leads to an error.

For ex.:

constexpr int x{2+3}; //ok

int y {};

cin>>y;

constexpr int d {y+2}; //error

This leads us to define CTC and RTC as follows

For RTC: use const directly

For CTC: use constexpr to make sure it is a CTC

* 1. Top-Level Const and Low-Level Const

TLC applies to the object, LLC applies to the type.

For ex.:

const int x{0}; //Top-Level

int\* const y{}; //Top-Level

const int\* z{}; //Low-Level

const int& m; //Low-Level

const int\* const n{}; //Left is LLC, right is TLC.

const int x means the value inside object x is constant, so its TLC.

int\* const y means the value inside z is constant, but since the value is an address, the value inside that address can change, however the address can’t change so it is still TLC.

const int\* z means the object being pointed to is const int, so it applies to the object being referenced, so it is LLC. Though the address can still change.

A simple way to understand this is to think in terms of the value (even the memory address value) at an address, if the value itself is constant that it can’t change then it is top level, but if the value is an address of another value and that other value can’t change then it is low level.

1. Scope

Variables declared inside a scope are only active throughout that scope. This is to say, that an object’s lifetime is limited to the scope it is present in.

For ex.:

void abc(int x)

{

//

}

void bb(int x)

{

//

}

abc(2);

bb(3);

abc and bb have different scopes and that is why the declaration of int x isn’t an issue nor does either x interject the other x.

In C++, objects start living from the point of declaration in the source code to the end of the scope, unlike Rust they aren’t deallocated at the last call site but only at the end of their scope.

int main()

{

// x can not be used here because it's not in scope yet

x=2;

int x{ 0 }; // x's lifetime begins here

doSomething(); // x is still alive during this function call

return 0;

} // x's lifetime ends here

In x=2, x is called ‘out of scope’.

Scopes flow one way, which is downwards. So identifiers defined on a higher level scope are available in the nested ones, but not the other way around.

* 1. Compound Statements

In C++, everything within {…} is considered a statement and this block is called a compound statement.

Blocks can be nested

int yo() {

int x{};

{

int y{};

std::cout<<x; //works

}

std::cout<<y; //doesn’t work due to scope rules.

}

This is called nesting depth, like here it is 2. Max. C++ compiler nesting depth is 256 but actual compilers support less.

Functions, classes, structs and the like require atleast 1 block which is both their definition and a compound statement.

But If, for loops and the like don’t require block so they can be ran without one too.

For ex.:

for (…)

std::cout<<”yo”;

works

* 1. Local variables

Basically the variables declared within a block. They are said to have “block scope” as their scope is limited by their block along with their storage duration which defines rules for their lifetime.

The storage duration is automatic for local variables as they are destroyed when the block ends automatically, this is why local vars are aka automatic variables.

int main()

{

int i { 5 }; // i created and initialized here

double d { 4.0 }; // d created and initialized here

return 0;

} // d and i are destroyed here

* + 1. Static Duration Local Vars

In C++ we can give static duration to local variables too, that is they are created at program start and destroyed at program end, their scope still remains the same. We use the static keyword for this.

For ex.:

#include <iostream>

void yo() {

static int me{ 0 };

++me;

std::cout << me << '\n';

}

int main()

{

static int me{ 6 };

std::cout << me << '\n';

yo();

yo();

yo();

std::cout << me << '\n';

return 0;

}

prints 6, then 1 then 2 then 3 and finally 6 again.

SLVs are only initialized the first time they are seen per scope and they are bound by their scope. So whilst yo() would see the same declared ‘me’ on consequent calls, main() would only see the ‘me’ defined for itself.

* 1. Global variable

It is advised to name non-const global vars with g\_ prefix.

These vars have file scope, aka global namespace scope or just global scope. That is, global vars’ scope is limited by the file they are in.

As for their duration, they have static duration as they are created at program start and only destroyed on program exit. Such variables are aka static variables. That is, before the main() the global variables and their constructors are called, and after main exits, these are then destroyed.

For ex.:

int g\_x;

const g\_y {1};

void pp(){

std::cout<<g\_x;

}

int main() {

pp();

}

works. Unlike local vars, global vars are zero initialized by default but they still need explicit initialization if const.

Non-const global variables should be avoided, as they can be modified by any file or piece of code and these side effects can become hard to debug.

* + 1. Sharing global vars across files

Prior to C++17 the way to do so is to simply define the global vars in a header and simply including it everywhere.

For ex.:

header.h

namespace constants {

constexpr int g\_x{2};

}

and in main.cpp

#include “header.h”

int main() {

g\_x; //usable

}

but this would copy g\_x into all cpp files the header is included, so if g\_x changes all the cpp files that use it would need to be recompiled.

Alternatively we can use RTC instead,

In header.h

namespace constants{

extern const int g\_x;

}

and header.cpp

namespace constants{

extern const int g\_x {2};

}

then use header.h in main.cpp like normal and the linker would link the value of g\_x to header.cpp.

The downside is that it won’t allow compiler to replace the occurences like constexpr would meaning these values would need to need a separate call.

C++17 solves this with inline variables,

Inline variables allow multiple definitions of the same identifier and at compile time pick one definition and use it as the main definition across all files.

For ex.:

In header.h

namespace constants {

inline constexpr int g\_x{2};

}

In main.cpp we use it as normal.

But any source file that wants to use g\_x would now need to meet 2 conditions, first it must have the identical definition and second it must give an explicit definition and not declaration for the identifier g\_x. (Since the header file simply gets copied into main.cpp we don’t need to redefine it).

And this way, if we have another file header.cpp that wants to define g\_x, it will have an identical definition like … g\_x {4}; and it will need to define it. Then main.cpp would pick header.h and then header.cpp would automatically become the primary definition used by main.cpp.

* 1. Shadowing

If a child scope declares variable with the same name as any of its parent scope then the variable ‘shadows’ the parent variables and all consequent calls within the same scope or deeper will modify this variable instead, unless we explicitly specify namespace (if any).

#include <iostream>

int x{ 0 };

namespace a {

int x{ 3 };

}

int main()

{

int x{ 2 };

{

std::cout << x << '\n';

std::cout << ::x << '\n';

std::cout << a::x << '\n';

}

return 0;

}

prints 2 then 0 then 3.

* 1. Linkage

Identifiers have another useful property called their linkage, which determines if other declarations of the same identifier refer to the same object or not.

Local vars have no linkage, meaning each declaration refers to a unique object.

For ex.:

{

int x{};

{

int x{};

}

}

Here the first x and second x both refer to unique objects and not the same.

Basically no linkage means the identifier is unique always.

* + 1. Internal Linkage

It means the identifier is only unique within the same file.

That also means, that it can only be accessed within that same file. And the identifier is not visible to the linker.

Global vars can have internal linkage, internal linkage global var is aka internal variable.

We use the static keyword for this.

For ex.:

static int g\_x{};

constexpr int g\_z{};

int g\_y{};

g\_x and g\_z have internal linkage meaning they are only accessible within this file, g\_y has external linkage. Non-const global vars have ext. linkage by default.

static can also be applied to functions to make them have IL.

IL doesn’t violate ODR as each definition is unique per file so it doesn’t affect other files.

* + 1. External Linkage

Identifiers with EL are visible to the linker and are hence unique per program. Can be accessed globally.

Functions have EL by default, this is why we are able to use functions from other files directly, we still need to forward declare them to tell the compiler to expect them though.

We can use the extern keyword to make identifiers have EL.

For ex.:

extern constexpr int g\_x{};

g\_x now has EL and can be accessed by other files.

But just like with functions, we need to tell the compiler to expect variable definitions from other files. To do so, we use variable forward declaration which is also enabled through extern keyword.

For ex.:

In add.cpp

extern const int g\_x{ 2 };

In main.cpp

extern const int g\_x;

and now g\_x can be used in main.cpp (given add.cpp is linked) and gets the value from add.cpp.

Yes this ambiguity is the reason why extern const global vars must have explicit initialization otherwise they are seen as forward declarations expecting a definition.

Constexpr can have EL but it is useless, because constexpr cannot be forward declared as the compiler expects a CTC value but that value isn’t available until linking phase. This is why we use const instead.

* 1. Scope Types

An identifier’s scope specifies where it can be accessed.

Block Scope / Local Scope: Can only be accessed within the same block (and nested blocks) from the point of declaration. Such as local variables, function parameters and type definitions inside blocks.

File Scope / Global Scope: Can be accessed anywhere from point of declaration to end of file. Such as global variables, functions, program-defined types.

Expression Scope: Expressions are assigned a scope as well, called the expression scope.

For ex.:

int add(int x) {

return x+2;

}

int main() {

add(2+3); //returns 7

}

Here we have 2 expressions, x+2 and 2+3, they both have expression scope. According to this scope, they are only accessible for the same expression.

The expression’s theirselves compute and go into an object, this object is called an Anonymous Object. An Anonymous Object has the expression scope, so it is created, computed, and destroyed in the same statement. So add(2+3) puts 2+3 in an anonymous object and then processes the value inside it, then sends it to add, similarly, return x+2 puts x+2 in an anonymous object, computes it, and returns the result.

* 1. Storage Duration

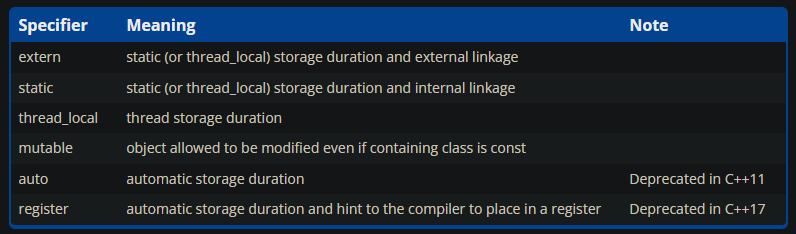
The storage duration of an identifier determines when it is created and destroyed, for ex. variables and function parameters are automatic duration so they are created at point of definition and destroyed when the block scope ends.

Static duration objects are created at program start and destroyed when program ends, for ex.: global vars and static local vars.

Dynamic duration objects are created at point of definition and destroyed when explicitly destroyed. Such as dynamically allocated variables

Storage Class Specifiers

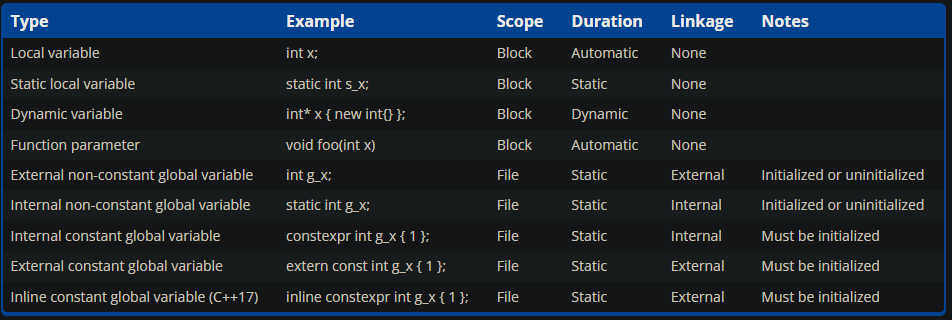
These specifiers define the storage duration and linkage of an identifier (not the scope).



The storage duration determines the lifetime of an identifier, the lifetime of an object is the time of an object from when it started existing to when it died. So an object with static duration has the biggest lifetime as it starts after program start and dies only at program end.

* 1. Linkage and Vars

Can be summarized like so



* 1. Lifetime
  2. Qualified vs. unqualifier scope resolution

Any identifier can be either qualified meaning we define the scope it is in. Unqualified means otherwise.

For ex.:

class C; // some class

C::s\_member;

obj.x;

ptr->y;

std::cout

::foo

are all qualified names. The class name qualifies s\_member, the class’ object qualifies x, a pointer qualifies y, std qualifies cout and global namespace qualifies foo.

* + 1. Using Declaration vs Directive

using declaration can be used to provide shortcut for a given identifier, whereas a using directive imports the names of a given namespace into an outer scope and allows current scope to use unqualified names to access these identifiers directly.

For ex.:

int main{

using std::cout;

cout<<”yo<<std::endl;

}

works, is a use of using declaration.

Using directives should be avoided as they could cause naming collisions, unintended shadowing and other hard to debug issues. Additionally, neither using directive nor using declaration can be cancelled/destroyed once initialized.

* 1. Function Scope

This scope is applies to an identifier and makes it visible throughout a function block even before they are declared.

Goto statement labels are an example of these.

1. Initialization
   1. Initialization of global vars

Happens in 2 phases, static initialization and dynamic initialization.

The first phase initializes all constexpr global vars with either default or given values.

The second phase initializes RTC and non const global vars.

It is inadvised to depend on RTC and non const global vars to initialize other global vars as the order of initialization (which file gets compiled first in an external linkage and etc.) isn’t fixed.

1. Multi-file programs

A single program can have multiple source files as well as header files.

To compile multiple source files using G++,

 g++ main.cpp add.cpp -o main

where main.cpp and add.cpp are 2 source files and main is the output executable name.

* 1. Definition

When multiple source files are used the compiler ‘forgets’ what declarations/definitions the file used after it is done compiling it. Then it moves to the next file and checks all of its definitions and goes on.

This is intentional and avoids scope pollution.

For ex.:

add.cpp has

int add ()

{

…

}

Then main.cpp has

int main() {

add();

}

This will give a compilation error as add is undefined for main.cpp.

The way to use identifiers from other source files is to use forward declarations,

so add.cpp remains the same but in main.cpp

int add();

int main() {

add();

}

Now this will execute the function in add.cpp as the compiler ‘remembers’ that it has to look for a definition to add();

Similarly, each source file has to #include its own headers and does not use another source file’s headers.

The only exception to not always needing a forward declaration is using header files. That is, if add is declared/defined in a header file then it can be directly used, not same for source files.

1. Namespaces

A namespace provides its own scope, which helps disambiguate identifiers.

For ex.:

Say we have 2 source files with both having a definition for

int add() {

…

}

Then, whilst this is not a compilation error, it is a linker error as we have a naming collision.

But if we use a different namespace in both files

namespace a {

int add () {…}

}

and in another file

namespace b {

int add() {…}

}

Then this will work. But inside a namespace the naming collision can still occur if multiple definitions are made inside a single namespace.

* 1. Nesting

Namespaces can nest.

namespace b {

namespace c{

…

}

}

is ok.

From C++17, this is allowed too

namespace a::b {

…

}

This means this block is defining stuff for namespace b which is inside namespace a.

* 1. Global namespace

Everything outside any namespace is implicitly included in the global namespace, also called the global scope. It is recommended to avoid polluting the global namespace.

To access identifiers in the global namespace we use :: without any prefix.

For ex.:

#include <iostream>

void pp() {

std::cout << "yo" << "\n";

}

namespace a {

namespace b {

void yo() {

::pp();

}

}

}

int main()

{

a::b::yo();

}

works. ::pp() means call pp in the global namespace.

* 1. std namespace

The standard library of C++ is a collection of functions, classes, constants etc. that provide a lot of functionality. Infact the standard document of C++ mostly has definitions for standard library’s members.

To avoid polluting the global namespace all of standard library’s members are present in the std namespace.

* 1. Using identifiers declared inside a namespace

To use identifiers declared inside a namespace we can either directly call them with

namespace a {

void pp () {

…

}

}

int main() {

a::pp();

}

The :: operator is known as the scope resolution operator and defines which scope to use (on the left) and what identifier to access (on the right). Yes, just like other identifiers even namespaces need to appear before main() in the global namespace.

or, we can use the using directive. This ‘opens’ the namespace for all the items below it. Or in other words, it allows access to all the items of the namespace without using the namespace prefix.

For ex.:

#include <iostream>

using namespace std;

int main()

{

cout << "Hello world!";

return 0;

}

And then all the members of std are available to be accessed without the prefix hence on in the global scope.

This is inadvised as it pollutes the scope and could lead to conflicts.

Using directive opens access for the scope it is in (and for the deeper ones) so if using has to be used, it must be used in the deepest scope, if it is used at all otherwise it pollutes the global scope.

* 1. Forward declaration with namespace

We can forward declare something in a namespace and define it using the scope resolution op.

For ex.:

#include <iostream>

namespace a {

void pp();

}

void a::pp() {

std::cout << "yo";

}

int main()

{

a::pp();

}

works.

* 1. Multiple same named namespaces

If multiple files and/or even same file has multiple namespaces with same name then they are all combined into one.

This means we can use namespace a in a header file to give forward declarations and its source file to give definition within same named namespace and use the header to simply call the method (considering the source file gets linked).

For ex.:

in add.h

namespace a{

void poo();

}

now in add.cpp

namespace a {

void poo(){

…

}

}

And finally in main.cpp

#include “add.h”

int main(){

a::poo();

}

Then, if add.cpp is linked then a::poo() will call the implementation defined in add.cpp

* 1. Namespace alias

We can use a different name for a namespace(s) by declaring it like a variable.

Syntax

namespace <name> = <namespace>:

and now <name> will call the namespace everywhere.

For ex.:

namespace a::b {

void poo(){…}

}

int main() {

namespace c= a::b;

c::poo();

}

works.

* 1. Unnamed or anonymous namespace

namespace without a name. All the content within an AN is available in its parent namespace(kind of like the namespace never existed), however it also applies the namespace benefits such as all its members have Internal Linkage only and have file scope.

For ex.:

namespace {

void pp() {

…

}

}

int main() {

pp();

}

works, as pp() is a part of its parent namespace, that is the global . But the function pp() only lives throughout this file as it has Internal Linkage. This is equivalent to static function for a function inside an AN.

* 1. Inline namespace

This namespace works kind of like the AN, the members are a part of the parent namespace. However, this namespace doesn’t apply Internal Linkage to all its members.

For ex.:

#include <iostream>

inline namespace b {

void pp() {

std::cout << "b" << '\n';

}

}

int main()

{

b::pp(); //also works

pp();

return 0;

}

works as pp() is part of the global namespace here. However pp() has no Internal Linkage so it can be picked up by other source files too.

This is useful for versioning things.

1. Headers

These files have a special extension that is .h or .hpp or sometimes no extension at all.

They allow us to give all declarations in them and then we can #include them in source files and use the identifiers directly without needing any forward declaration.

For ex.:

#include <iostream>

int main()

{

std::cout << "Hello, world!";

return 0;

}

Here, cout is not defined for this file yet this works as it is defined in the iostream file and PPD copies the content of iostream over into this file.

* 1. Usage

Header files are meant to only have declarations and not definitions as they may void ODR if they have definitions.

Ideally, we must pair a source file with a header file. The header file provides the declarations and the source file provides the definitions.

For ex.:

add.h

int add(int x, int y);

Then in add.cpp

#include "add.h"

int add(int x, int y)

{

return x + y;

}

and finally in main.cpp

#include "add.h" // Insert contents of add.h at this point. Note use of double quotes here.

#include <iostream>

int main()

{

std::cout << "The sum of 3 and 4 is " << add(3, 4) << '\n';

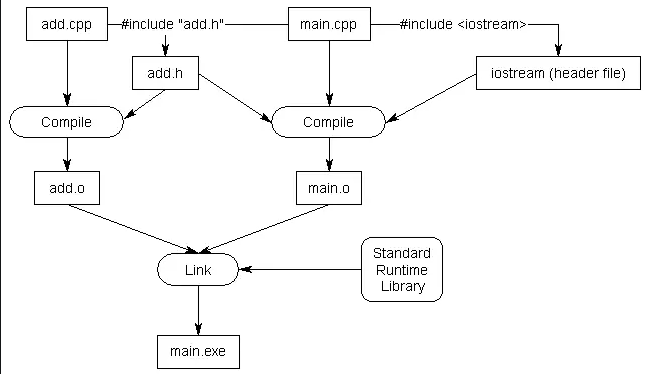
return 0;

}

User-defined header files are #included with double quotes. This is to indicate that the header files are in the project directory and not the include directories (directories where OS, compiler or third party libraries are stored). If the preprocessor doesn’t find file under “” then it looks in the include directories. <> headers are only looked for in the include directories.

Here, there’s nothing new happening. The add.h header declares the function add, which in the TU appears as-is. Then the compiler compiles the add.cpp and main.cpp and uses the definition in add.cpp for the function add.

Visually,



The standard runtime library means C++ standard library here.

The reason add.cpp also #include’s add.h is to allow the linter to check for syntax at the time of programming. It’s not required to do so but it ensures the linker doesn’t complain later about an incorrect definition.

* 1. Why isn’t iostream iostream.h ?

iostream used to be .h but later its contents were moved into the std namespace and a new header iostream was introduced. This is also to say that headers without .h open their contents in the global namespace or std namespace generally.

* 1. Include from other directories

Header files from other directories can be #included as well.

#include "headers/myHeader.h"

works but is inadvised. It is recommended to define such directories in include directories.

For ex.:

In G++ we can use this

g++ -o main -I/myfolder/headers/ main.cpp

adds the given dir to the included directory.

* 1. Transitive include

Headers may #include other headers, and the preprocessor will open those headers into the current file as well. Meaning if a header a.h includes b.h then in the main.cpp if we include a.h then b.h is implicitly included as well.

This is why iostream or other std libs may implicitly also allow usage of some other header.

It is recommended to explicitly include all required headers and not rely on transitive includes.

* 1. Include order of headers matter

The earlier in the code a header is #included the earlier its contents opened. Meaning if a header A depends on another header B but doesn’t include it then it’s an error, still if we include B it in our main then A then this works as it is equivalent to A including B (as the same order will appear in the TU) but if we include A then B then it will be an error.

This is why it is recommended to include in this order

1. The paired header (the header being implemented in this source file)
2. Other project headers
3. 3rd party headers
4. STD headers
   1. Header Guards

A header can also define identifiers. This will cause error if multiple headers define the same identifier and all of them are #included somewhere.

A header guard is just a technique to use PPDs to avoid such a situation. To do so,

#ifndef SOME\_UNIQUE\_NAME\_HERE

#define SOME\_UNIQUE\_NAME\_HERE

// your declarations (and certain types of definitions) here

#endif

And all the contents of the header go between the ifndef and endif.

The way this works is simple, if the macro is undefined then define the macro and then the header, but if it is already defined, i.e., another header that does the same thing is already included or the same header is transitively already included in the including file then don’t define the header.

Another simpler HG is

#pragma once

Writing this at the top of a file does the same as traditional HG, however #pragma is not a part of the C++ standard and is not guaranteed to always be supported, furthermore there are a few cases where it could fail to do what it’s intended to do.

1. Type Conversion

C++ implicitly converts values of some types to other types but custom types and some base types require explicit conversion to be both defined and used.

The implicit conversion also throws a warning if the type will lose some data upon conversion.

When value of a type is converted into value of another type then it is called type coercion.

For ex.

double d{5.0};

int c{d};

might work fine and it won’t even lose the value but it is still a potential data loss. (It might work fine in other contexts but in brace initialization, this will always be an error). For ex. 5.5 to int would only put 5 in int c.

However, int to double will never have a data loss so it is not a warning.

* 1. Numeric Promotion

Depending on the CPU, it could be faster for a CPU to process 32 bit types than smaller ones. So these narrower (in width, width is the no. of bits of a data type) types are automatically promoted to a wider type automatically by the compiler, this is called numeric promotion.

These preserve the value as the range simply increases and not decreases so there’s no chance for data loss hence it is not a warning. However, numeric promotion is not signedness preserving so a signed type might be promoted into an unsigned type.

Floating point promotions are when floats are promoted to double automatically.

Integer promotions are a bit nuanced. Here’s how they are promoted

<https://en.cppreference.com/w/cpp/language/implicit_conversion#Integral_promotion>

* 1. Numeric Conversion

When a numeric value is converted to another type, either implicitly or explicity (using casting).

A narrowing conversion is one where a wider type is converted into a narrower type.

For ex.:

long x{2};

int y {static\_cast<int>(x)};

works.

* + 1. Implicit Type Coercion

aka Automatic Coercion happens between most fundamental types.

Such as

int x{0};

if (x) {…}

Here x is converted to bool where 0 means false.

Brace initialization throws an error if an ITC results in data loss.

For ex.

int x{2.2}; //error

Some types need 0 conversions to be of another type, such as int to long or vice versa.

* + 1. Arithmetic Conversion

When operands in an operation are

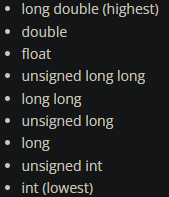
1. either not the type that the operation supports but it is an integral type and it can be promoted
2. Or not the same types,

Then ‘usual arithmetic conversion’ rules are applied to them which basically implicitly convert the operands to the same type according to priority list.

For 1, they are both promoted to the same type in the list.

For 2, the type that is lower level in the priority list is converted to the higher priority type.

This is the priority list



As we can see mixing signed and unsigned operands can give odd results as unsigned has higher priority.

* 1. Casting

We can force casting a value into a value of another type, called explicit conversion.

To do so we make use of casting in c++ which is done through some special operators.

There are 5 types of casts in c++,

C-style casts, static casts, const casts, dynamic casts and reinterpret casts.

* + 1. C-style cast

Type is defined inside () and converts the value ahead into this type.

For ex.:

double d{ (double)2/4 };

double e {double(2)/4};

works. Stores 0.5 in d and e. Both are valid syntax.

C-style cast uses other casts and automatically picks one based on context. So it could use static cast at one place or reinterpret cast somewhere else. This is also why it should be avoided as it may cast using an unwanted cast.

* + 1. Static Cast

Simply converts values.

To use it

static\_cast<new type>(<some value>);

For ex.:

static\_cast<int>(5.0);

will return 5 and will never be an error because we explicitly define this conversion.

This is the most basic type of explicit conversion in C++ and doesn’t offer any range checking (checking if value is holdable by the given type) and gives unpredictable results when forced to convert a value that can’t be directly converted such as a signed int -2 to unsigned int.

* + 1. const cast

This cast adds/removes constness from a pointer/reference type.

For ex.:

int x{50};

const int\* y{&x}; //now we can’t change value of x.

int\* z{const\_cast<int\*>(y)}; //cast y as int\*

\*z=10; //and now x is changed to 10

or we can cast int\* as const int\*.

1. Type Aliases

These are used to define alternate names for types.

We have using and typedef as 2 ways to do so in C++.

For ex.:

using D= double;

//now

D x {2.0}; //works

This is much more useful when working with generics.

It is recommended to name alias types PascalCase.

These custom type values can be used interchangeably with their aliasing type values.

Type alias have the same scoping rules applied to them as variable identifiers. So

using D inside a block will limit the scope of the alias to that block.

* 1. Typedef

Older way of defining a type alias.

typedef Double D;

works and is identical to the modern type alias.

Complex type aliases get confusing fast with typedefs,

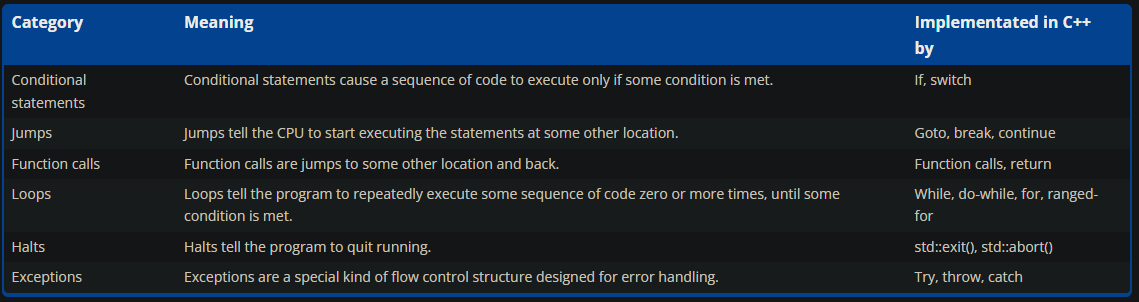
typedef int (\*FcnType)(double, char);

using FcnType = int(\*)(double, char);

Both are the same but modern one is more simpler imo.

1. Conditionals

C++ supports these control flow statements



* 1. If-else if- else

Work like other langs.

They require blocks but if we don’t specify then the next line is implicitly taken as a block.

For single line this form is advised

if (true) dosomething();

; is considered a statement end and is valid for a single line function, so

if (true)

;

dosomething()

Here the second line is taken as a statement hence the if block is empty and dosomething() is called regardless of condition.

* 1. Switch

For ex.:

void printDigitName(int x)

{

switch (x)

{

case 1:

std::cout << "One";

break;

case 2:

std::cout << "Two";

break;

case 3:

std::cout << "Three";

break;

default:

std::cout << "Unknown";

break;

}

}

The expr. inside switch could be anything but it must return an integral type or an enumerated type, this is due to how switch works. It is implemented as an optimized conditional statement so it must be used for >=5 conditions for optimal performance.

Yes chars are also integral type.

Each case arm is called a case label and lastly we have the default label.

Finally the break is required unless we want the execution to continue executing other case label’s bodies. Yes, if a single case label matches and it doesn’t break then the execution will fall through into the next case without condition checking and run its block and so on until it breaks or the switch gets finished or returns.

For ex.:

switch (2)

{

case 1: // Does not match

std::cout << 1 << '\n'; // Skipped

case 2: // Match!

std::cout << 2 << '\n'; // Execution begins here

case 3:

std::cout << 3 << '\n'; // This is also executed

case 4:

std::cout << 4 << '\n'; // This is also executed

[[fallthrough]];

default:

std::cout << 5 << '\n'; // This is also executed

}

prints 2 then 3 then 4 then 5.

This is called fallthrough and compilers flag it as a warning. We can let the compiler know it is intentional with [[fallthrough]] attribute.

switch (c)

{

case 'a':

case 'e':

case 'i':

case 'o':

case 'u':

case 'A':

case 'E':

case 'I':

case 'O':

case 'U':

return true;

default:

return false;

}

This is not considered a fallthrough because all the case labels share the same body, that is the last block with return true.

The reason fallthrough is a thing is because case labels are simply labels and the condition is checked only until the first one matches. The case bodies are still inside the switch block.

That is,

  int x{ 2 };

switch (x) {

case 1:

int y;

y = 2;

std::cout << "1" << '\n';

case 2:

std::cout << "2" << '\n';

std::cout << y << '\n';

default:

std::cout << "default" << '\n';

break;

}

works. Though y has garbage value in it as it does not get the value defined in case 1.

Variable declarations & definitions inside switch and case labels are ok, but they can’t be initialized, however they can be assigned values. They can be initialized if inside case or default label block.

For ex.:

int x{ 2 };

switch (x) {

int z{ 2 }; //error

case 1:

int y;

y = 2;

std::cout << "1" << '\n';

case 2: {

std::cout << "2" << '\n';

std::cout << y << '\n';

int m{ 12 }; //ok

}

default:

std::cout << "default" << '\n';

break;

}

* 1. Goto

Unconditional jump, that is the jump always happens and there is no condition.

To use goto,

int main() {

int x{2};

yo:

std::cout<<”yo”<<’\n’;

if (x==2){

++x;

goto yo;

}

}

works. yo is called a statement label here. It must have a statement after it, even empty statement ; works.

statement labels have function scope. So,

void yo() {

if(true) goto yo;

std::cout<<”na”;

yo:

;

}

works.

Goto should be avoided because they make the execution path harder to grasp and debug.

* 1. Break and Continue

Break is used to break out of an immediate loop and continue takes the control to the end of the loop block.

Break also exits out of switch case labels.

* 1. Halts

A function that terminates the program/thread in C++. It is inadvised to call halt explicitly.

* + 1. std::exit()

Terminates the program normally after performing some cleanup statements automatically. It is implicitly invoked after main() has returned.

We can explicitly call it with

#include<cstdlib>

int main() {

int x{2};

std::exit(0); //terminates and sends 0 to OS

int y{3};

return 0;

}

The lines after exit() aren’t ran and it runs some cleanup statements then terminates the program.

However it doesn’t cleanup current scope members when called explicitly so it should be avoided.

* + 1. std::atexit()

Takes a void parameterless function as arg and then runs it when exit is invoked.

For ex.:

void yo() {

…

}

void pp() {

…

}

int main() {

std::atexit(yo);

std::atexit(pp);

…

}

Multiple functions can be assigned to atexit and they are called in reverse order of registration. So pp() gets called first then yo() here. These are invoked whenever exit() is invoked and not before.

* + 1. Quick Exit

In multi-threaded programs, calling exit() explicitly may crash the program as it will clean static objects which may be accessed by other threads.

That’s why C++ also has std::quick\_exit() and std::at\_quick\_exit(), these functions don’t perform any cleanup and simply terminate a thread/program.

* + 1. Abort and Terminate

std::abort() terminates the program/thread without any cleanup and signals the OS that something went wrong with the program.

std::terminate() is called when exceptions aren’t handled and it implicitly calls abort().

1. Loops

The usual

while (<condition is true>)

{

…

}

do

{

…

}

while(<condition is true>);

and

for(<init var>; <condition is true>; <inc/dec>)

{

…

}

for(;;){…} is equivalent to while(true){…}

For multiple initializations in for

for(int x{},y{};…;…){…}

One thing to note is that loops should rely on signed ints for their condition to avoid mean int overflow/underflow bugs.

We can also use this for loop to traverse arrays,

std::array arr{1,2,3};

for(auto i{arr.size()}; i-- > 0;) {…}

This loop traverses arr in reverse, and whilst normally we’d have size()-1 (which would need casting as 1 is of type int and size() returns size\_t), we do the decrement with the check, so it checks i > 0, then decrements i and repeats this for every iteration (It does underflow but that doesn’t matter, if size is 0 or i is 0, then i > 0 would fail, then i— would wrap around i to maximum value, but the loop would have already ended).

* 1. Range-Based for loops

Allows directly iterating over arrays.

For ex.:

constexpr int arr[ ] {1,2,3,4};

for(int elem: arr) {

…

}

Iterates over each element of arr and copies it to elem. We can also use references to avoid copying and directly modifying the array element.

for(int& elem: arr) {…}

The type can be auto as well.

Similarly, const can be used to disallow modification.

For ex.:

for(const auto& elem: arr) {…}

It is recommended to only use references for types that take >4bytes as just like references in functions, it implicitly passes memory addresses which take space.

However RBFL doesn’t work for T\* of arrays.

There’s no way for RBFLs to have a counter for keeping track of index (we can do so externally) until C++20, from C++20 we can also provide an init-statement.

for(init-statement; element-declaration: array)

For ex.:

int arr[5] {1,2,3,4,5};

for(int index{0}; const auto& elem: arr) {

…

++index;

}//works

RBFLs actually call the begin() and end() methods of the type, in the above case they use some predefined methods for int[] but for other types they call their begin() and end() methods, this is how they interally create a loop from begin() to end().

1. Error Handling

Generally errors should be handled in 4 ways,

1. Handle the error at the site
2. Pass the error back to a caller or up the stack
3. Halt the Program
4. Throw an exception

2 simply means indicating the caller of a failure, generally through Boolean return.

3 means using one of the halt methods

* 1. Indicating an error

If we know a given value will give error we can use std::cerr to output text to error stream.

For ex.:

if (x>2)

{ … }

else

{

std::cerr<<”Yo failure!”;

}

* 1. Assertion

This is an expression that checks if a statement is true. If it isn’t, then it aborts the program.

In C++, runtime assertions are defined using the assert macro.

For ex.:

#include<cassert>

void pp(int x)

{

assert(x==2);

…

}

int main()

{

pp(4);

}

would abort as soon as that assert fails.

Prints

Assertion failed: x==2

Using a neat trick we can even get it to print error strings.

assert(expr && <someString>);

prints

Assertion failed: expr && <somestring>

This works because string literals always return true with Booleans and that means expr has to return true to fail.

asserts should only be used in debug builds and not prod builds as they incur a small performance penalty each time they are checked. We can use NDEBUG to disable all asserts, this is defined to the compiler directly.

* 1. Compile Time Asserts

These asserts are only ran at compile time. C++ defines static\_assert to achieve this.

For ex.:

int main() {

static\_assert(2==8, “Failed”);

}

works.

1. Random

C++ provides a good number of algs and methods to work with random numbers and generating sequences of them. Whilst the alg to generate them are bad, except Mersenne twister which is just ok, there are better external libs such as wyhash.

To use MT,

#include<random>

#include<iostream>

int main() {

std::mt19937\_64 mt{};

std::cout<<mt()<<’\n’;

std::cout<<mt()<<’\n’;

}

works. mt generates 64 bit unsigned random int each time it is invoked. There’s also 32 bit version without the \_64.

* 1. Distribution

By default the PRNGs (Pseudo Random Number Generators) have a fixed range of values, like between 0 to some large number. We cannot change that but we can use a distribution function to map that range into a smaller range.

For ex.:

std::mt19937\_64 mt{};  
std::uniform\_int\_distribution<> die6{ 1, 6 };

std::cout<< die6(mt);

Uniform int distribution is one such distribution that takes a range and then maps the output of a bigger range and returns 1 number from its range. There’s equal probability for all numbers in the given range to be picked for any given input in UID.

* 1. Seeding

By default the seed for PRNGs in C++ is fixed, so they generate the same numbers everytime they are ran. To fix this we are required to seed with a random value, time is the best natural random value we have so we can use C++ chrono library for it.

For ex.:

#include<chrono>

#include<random>

int main() {

std::mt19937 mt{ static\_cast<unsigned int>(std::chrono::steady\_clock::now().time\_since\_epoch().count()) };

}

This gives the current time in milliseconds or nanoseconds since epoch.

We can alternatively use high\_resolution\_clock() instead of steady\_clock() but it depends on system time, which can be manually adjusted.

We can also use

std::random\_device{}()

This gets a random value from the OS and can be used to seed the PRNG. We don’t use this itself for the PRNG as it may be expensive and it’s performance and quality is implementation defined.

PRNGs require a fixed seed size, for example MT requires 624 bytes to be properly seeded. We only provided 4 bytes, and then the other 620 bytes are chosen automatically which may lead to poor PRNG, this is called underseeding.

We can provide multiple values to PRNGs to fill up the seed, additionaly we can use std::seed\_sq to both hold multiple seed values and also fill in the rest with a bit better quality random values.

To use it

std::random\_device rd{};

std::seed\_sq ss {rd(),rd(),rd(),rd(),rd(),rd(),rd(),rd()};

std::mt19937\_64 mt{ss};

works and gives 8 values to seed\_sq, which generates the rest 616 bytes and passes them to mt. We don’t generate 600+ rd() as it could incur a huge performance penalty. seed\_sq generates better remaining values the more values are given to it.

Warming up: It is optimal to generate the first random number of values by a PRNG and discard them to mix its internal state up. Seed\_sq does this automatically.

1. Template

A template in C++ is a definition that allows the programmer to define a blueprint of an object which the compiler then uses to generate proper definitions at compilation.

That is, we define a template function/class etc. and it uses placeholder types and performs some operations, then when we call the template function we define what the type is or the compiler automatically infers it and the function/class etc. works normally. At compile time, the templates are replaced and proper functions/classes etc. defined that only accept the type used at call-sites so the result is identical to as if we defined those definitions by ourself except the compiler generates them automatically, yes this means if a template has calls from different types then all those types are also defined automatically by the compiler.

The benefit provided by templates is that if we have many operations that perform the same thing but accept multiple types then we only need to define a template and once instead of manually defining each normal function/class explicitly.

For ex.:

template<typename T>

T add(T x, T y) {

return x+y;

}

add<int>(2,3); //works

add<double>(2.0,3.0);

add(5,6);//works

Here template itself is declared with template keyword and T is the placeholder type. The compiler generates 2 add functions, one with int and one with double where this type replaces T wherever it is used.

The parameter to the template<…> definition is called a template parameter, there are 3 kinds of these

1. Type Template Parameter (where the param is a type, for ex.: typename T)
2. Non-type template Parameter (param is a constexpr value)
3. Template template Parameter (param is a template)

It is the standard to use single capital letters to represent placeholder types.

* 1. Function Template

Templates defined for a function. The compiler generates the overloads for the template function.

For ex.:

template<typename T>

T add(T x, T y) {

return x+y;

}

And called with add<type>(<value of that type>, <another value>).

The <type> is called the template argument.

Function templates aren’t functions theirselves, just a blueprint for actual function that will be generated.

If we call add<int>(2,3); then this will be generated

template<typename T>

T add(T x, T y) {

return x + y;

}

template<>

int add<int>(int x, int y) {

return x + y;

}

int main()

{

std::cout << add<int>(2, 3) << '\n';

}

(Yes this is actual code here, it is identical to what compiler will generate but this is called explicit function template instantiation).

The process of creating the appropriate functions for templates is called function template instantion and when it happens at compile time due to a call, such as add<T>(…) then it is called an implicit function template instantiation. This instantiated function is called a function instance.

Obviously if we have a function template that uses T in a way that can’t be used then this will result in an error.

For ex.

template<typename T>

T add(T x) {

return x+1;

}

and we call add<std::string>(“yoooo”); then this will be an error. Fortunately, a compile time error.

Templates types are aka Generic Types as a ‘generic’ type can map to a lot of other types.

* + 1. Template Argument and Overload Resolution

Template arguments also define the overload resolution’s result, so

template<typename T>

T add(T x) {

return x+1;

}

Here, if we call add<double>(2,3);, it would call the function instance where double is used.

* + 1. Template Argument Deduction

We actually don’t need to define the type to the template argument, add(2,3) would also work. This is because of Template Argument Deduction (different from CTAD) which deduces the type T based on the parameters given.

add(2,3); and add<>(2,3); both do the same thing. However, this doesn’t define the overload resolution’s result so if we wanted to call add<double>(…) but the params were int and we used add(…) then it would call the int instance.

TAD doesn’t do type conversion. So,

template<typename T>

T add(T x, T y) {

return x + y;

}

Here if we call add(2,3.5); then this would be an error as type is int,double for params whereas the function template requires both types to be the same.

However, add<int>(2,3.5); is not an error as TAD isn’t applied and both params are coerced into T, which is int here.

Alternatively we can just use multiple types in the template parameter and function signature.

For ex.:

template<typename T, typename U >

auto add(T x, U y) {

return x + y;

}

and add(2,3.5); works now as TAD can see 2 types for function template of add. T and U could be the same or distinct, there’s no relation between them

We can use ‘auto’ as a return type to functions, this deduces the correct return type automatically.

* + 1. Non-type template param function template

Basically non-placeholder types when used in template functions are treated normally so

template<typename T>

T yo(T x, double y) {

return x+y;

}

works and y is called a non-type template param.

We can also use non-type template param in the param itself

#include<iostream>

template<typename T, int N>

void yo(T k[N]){

std::cout<<N<<'\n';

}

int main() {

int x[10]{10};

yo<int,10>(x);

}

works, the int is passed value with a constant.

* + 1. Forward Declarations need definitions

Function templates need a body at compile time to generate actual functions so forward declarations are not ok if a source file doesn’t define one.

template<typename T>

T yo(T x, double y);

and directly using yo(…) is obviously an error, but if this yo was in a header then each source file would need to define a body for it otherwise it would be an error if they used yo() with a type that hasn’t been instantiated by a source file that does define yo().

That is,

Say we have 4 files,

x.h

template<typename T>

T yo(T x, T y);

x.cpp

#include “x.h”

#include<iostream>

void callYoX(){

std::cout<< yo(2,3) << ‘\n’;

}

Then y.cpp

#include “x.h”

#include<iostream>

template<typename T>

T yo(T x, T y) {

return x+y;

}

void callYoY(){

std::cout<< yo(5,10) << ‘\n’;

}

and main.cpp

#include<iostream>

void callYoX();

void callYoY();

int main() {

callYoX();

callYoY();

}

Then this works despite x.cpp not having any definition for yo(). This is due to the linker which links the int instantiated yo() to callYoX(). Compiler didn’t generate any instantiation for x.cpp, it is the linker that simply links the one from y.cpp.

However if callYoX() calls, say double instantiated yo(), then this is a linker error as x.cpp doesn’t have the definition and compiler couldn’t generate any intantiation.

To avoid this situation, it is recommended to simply give definition in the header itself and if the source file needs a different definition it can define that too, as Function Templates don’t follow ODR.

* + 1. Abbreviated Function Template

Brought with C++20, it is a shorthand for Function Templates. We use auto to use this feature.

For ex.

auto add(auto x, auto y) {

return x+y;

}

is a shorthand of

template<typename T, typename U>

auto add(T x, U y) {…}

* + 1. Template parameters can have default arguments

template<typename T =int>

T add(T a, T b) {…}

int main() {

add(1,2); //T is taken as int

}

* + 1. Function Template Specialization

Aka full/explicit function template specialization.

Allows us to define a custom instantiation for a template type.

That is, for a given template instantiation we can give a special definition for the function.

To do so,

#include <iostream>

template<typename T>

void yo(T a) {}

template<>

void yo(int a) {}

int main() {

yo(2);

}

here template<> is used to give an EFTS for yo() when T is int. yo(2) then calls our EFTS since T is int.

When the compiler discovers a call to yo<int>() and it already knows there is an instantiation for yo<int>() explicitly defined then it uses that instead of creating a new one from the template definition.

* 1. Parameter Pack

This generic type accepts 0 or more args, if a template that accepts a parameter pack is called a variadic template. We use ellipsis here too but the difference is that they are strongly typed.

For ex.:

template<typename… Ts>

void yo(Ts… args) {

…

}

int main() {

yo<int>(1); //ok

yo(); //ok

yo(3); //ok, CTAD from C++17 deduces the type as int

yo<double,char>(4.0,’a’); //ok

yo<0>(); //error, 0 is not a type

}

and args contain the variable number of args.

For functions, the parameter pack in template parameters could appear anywhere, at the front or back etc. But for structs it must always be the last parameter. And for function parameters, it must always be the last parameter too.

For ex.:

template<typename… Ts, typename U>

void yo(U u, Ts… args); //ok

//calling yo(1.0,2,3,’a’); means U is of type double, Ts have args of types {int,int,char}

template<typename… Ts, typename U>

void yo2(Ts… args, U u); //error

template<typename… Ts, typename U>

struct X{…}; //error

template<typename U, typename… Ts >

struct X2{…}; //ok

This is why it is recommended to always have the parameter pack at the end of template parameter and also for function parameters to avoid confusion.

Pack expansion

The ellipsis operator expands the contents of the parameter pack in-place. It needs 2 things, pattern and the ellipsis to indicate pack expansion.

For ex.:

template<typename… Ts>

void yoyo(Ts… args2){…}

template<typename… Ts >

void yo(Ts… args){ //pack expansions starts from here itself, the pattern here is Ts…, it expands to double E1, const char\* E2

args…; //args is the pattern, so it expands to E1, E2

&args…; //expands to & E1, &E2 (so E2’s type is const char\*& here), if we pass references to templates, they turn to pointers

yoyo(&args…); //sends references, this makes Ts… args2 in yoyo as double\* E1, const char\*\* E2

}

int main() {

yo(2.0,”yoo”);

}

That is, the pattern dictates what kind of expansion will occur, Ts… means the types have to be expanded as-is, args…. means the values have to be expanded as-is, &args… means the references to values have to be expanded. Similarly, Ts…& means types have to be expanded as references, so void(Ts… &args) means double& E1, const char\*& E2.

Here, E1 and E2 are just placeholder names, the actual names are upto the compiler and can’t be accessed directly anyway.

If multiple parameter packs are being expanded in the same statement, then they must have same no. of arguments.

That is,

#include<vector>

#include<utility> //defines pair<T,K>

template<typename… Ts>

struct X{

template<typename… Us>

struct Y {

typedef std::vector<std::pair<Ts,Us>…> myType;

//std::pair<T,K> is the pattern here and will be expanded

};

};

int main() {

typedef X<int>::Y<char>::myType T1;

//causes myType’s typedef to expand to std::vector<std::pair<int,char>>

typedef X<int, char>::Y<float, double>::myType T1;

//causes myType’s typedef to expand to std::vector<std::pair<int,float>, std::pair<char,double>>

typedef X<int>::Y<float, double>::myType T1;

//error

}

If 2 parameter packs expansions appear in the same statement and 1 is nested by the other, then the innermost one is expanded first.

For ex.:

template<typename… Ts>

void yo(Ts… args) {

const\_cast<const Ts\*>(&args)…;

//cast each arg’s reference as const Type\*. Here the pattern is const\_cast<const Ts\*>(&args).

f(h(args…)+ args…);

//Here innermost pack is args… so it is expanded first to make h(E1,E2) and pattern was args…, then the outermost pack is +args… where pattern is +args so it makes f(h(E1,E2)+E1, h(E1,E2)+E2);

}

Depending on where the expansion occurs, the resulting comma-seperated list is different.

That is,

For 3 values in the args

When used in function call expands to args,

f(args…); //expands to f(E1, E2, E3)

f(&args…); //expands to f(&E1, &E2, &E3)

f(n, ++args…); //expands to f(n, ++E1, ++E2, ++E3)

f(++args…, n); //expands to f(++E1, ++E2, ++E3, n)

f(h(args…)+ args…); //expands to f(h(E1,E2,E3)+E1,…)

When used in function params expands to types,

template<typename… Ts>

void yo(Ts… args) {…}

template<typename… Ts, int… N>

void yo2(Ts (&…args)[N]) {…}

int main(){

yo(1,’a’); //makes yo expand to void yo(int E1, char E2)

int n[3];

yo2(“a”, n); //makes yo2 expand to const char (&E1)[2], int &E2[3], E1 is “a\0” that’s why size 2

}

parameter pack can be made out of non-type template parameters too, in this case we can see args expand and each arg is passed its value for N.

When used in initializer list, same as function call

SomeClass c1(args…); //expands and calls SomeClass::SomeClass(E1,E2,E3); the ctor that accepts these 3.

When used in expressions as Brace-enclosed initializers (inside braces, i.e., any kind of brackets),

const int size= sizeof…(args) +2; //sizeof… is an operator made specifically for getting no. of args from parameter pack.

int array[size] {1, args…, 2}; //works

int anotherArray[sizeof…(Ts)] = {(std::cout<<args,0)…};

//Here (…) is the pattern so it is repeated with args replaced with each arg, the comma operator returns the right operand so 0 is returned for all expressions.

When used as type arguments,

template<typename… B>

void yoyo(B… b) {…}

template<typename A, typename… B>

void yo(A a, B… args) {

yoyo<A, args…>(a, args…); //expands to yoyo<A, int,int,int>(‘a’,1,2,3);

yoyo<args…, A>(args…, a); //works

}

int main() {

yo(‘a’, 1,2,3);

}

It automatically expands the types instead of values when they are required.

When used as parameter to another template,

template<typename… Ts>

struct X{

template<T… values> //expands to non-type template params, like template<int, char>

struct Y{…};

};

When used with classes

template<class... T>

class X : public T...

{

public:

X(const T&... mixins) : T(mixins)... {}

};

Initializes class definition with given no. of types as the class extends all of them, then the ctor accepts values of all these types and for each type calls the base ctor for that type.

Can be used in lambda capture too.

1. Memory

The stuff related with how a program uses the memory.

* 1. Segments

Memory used by any program is divided into different areas, called segments. There’s

Code/Text Segment: Read-Only segment with compiled code only.

BSS Segment: Zero-initialized global and static vars are stored here.

Data Segment: Initialized global and static vars are here.

Heap Segment: Dynamically Allocated variables get their address from this space.

Call Stack Segment: Function Params, Local Vars and the like are here.

* + 1. Heap Segment

Slower than Call Stack Segment for allocating memory. Dereferencing is also slower than direct access to call stack variables. Big pool of memory, so for large space requirements heap’s use is recommended.

* + 1. Call Stack Segment

Commonly called just the stack. Keeps all the locally required values. Is implemented as a stack data structure (LIFO data structure). Has very limited size.

So, when a C++ program starts, the main() function is pushed to the stack, then it defines variables etc. which are inserted into the stack as well, then any functions are inserted etc. and each occupy a space, each of these item is called a stack frame and the stack keeps track of its top with a ‘marker’, which is a register in the CPU that holds this info and the marker is aka Stack Pointer (SP). When something goes out of scope/lifetime such as a function ending, the marker walks back and all the local vars etc. stack frames are sequentially ignored until all the stack frames until the function’s call site are ignored, this process is also known as stack unwinding, and now the main function continues and any new item overwrites the pre-existing item in the stack, all thanks to the marker. As for the memory allocated by items, it is cleared in the unwinding.

Each stack frame consists of these items:

* The address of the instruction after the function call (so that the stack can return to this instruction and continue this function).
* All function args
* Memory for any local vars
* Saved copies of all the registers that are being modified by the function to restore them during unwinding.

This is why deep recursion is affected by the number of arguments passed to the function.

For ex.:

int foo(int x)

{

// b

return x;

} // foo is popped off the call stack here

int main()

{

// a

foo(5); // foo is pushed on the call stack here

// c

return 0;

}

At a we have

main()

in the call stack, at b we have

foo() (and param x)

main()

in the call stack and at c we have

main()

The stack is severely limited in memory, 1 MB by default for Visual Studio, 8 MB for G++.

This means if we have a really deep function call chain or an object that takes up a larger amount of space then it results in an infamous error, the stack overflow error which leads to program crash.

For ex.:

#include <iostream>

int main()

{

int stack[10000000];

std::cout << "hi" << stack[0]; // we'll use stack[0] here so the compiler won't optimize the array away

return 0;

}

Allocates 40MB on the call stack, since it is so huge it leads to stack overflow.

And this snippet shows how function call chain leads to stack overflow

#include <iostream>

int g\_counter{ 0 };

void eatStack()

{

std::cout << ++g\_counter << ' ';

// We use a conditional here to avoid compiler warnings about infinite recursion

if (g\_counter > 0)

eatStack(); // note that eatStack() calls itself

// Needed to prevent compiler from doing tail-call optimization

std::cout << "hi";

}

int main()

{

eatStack();

return 0;

}

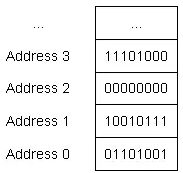
Both crash the program and the program exits with a code -1073741571 on windows, this code is special and indicates access violation.

So, call stack is small, fast, memory allocation is known at compile-time, scope is auto managed and actual (de)allocation of memory is fast too.

1. Data Types

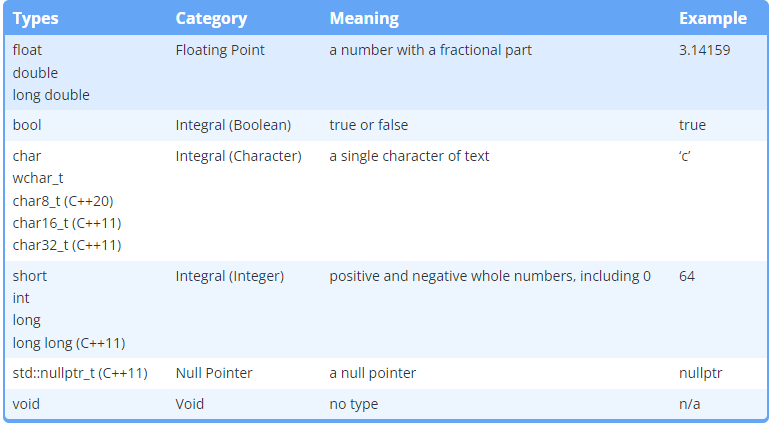
Smallest unit in a computer is a bit, which can be 0 or 1. 8 bits make a byte.

A byte is what a memory address stores on the RAM

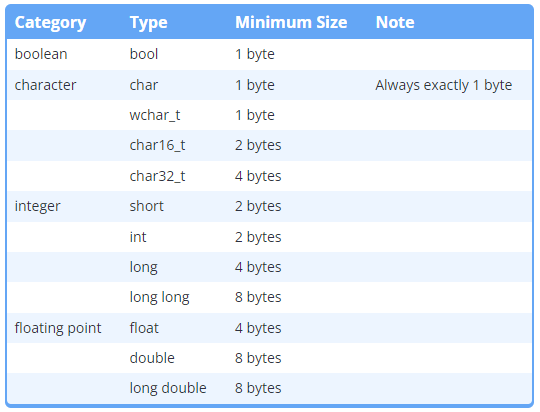


That is to say, all data types, all data in any language is basically represented with bits.

* 1. C++ Fundamental Data Types



whilst they have the sizes



2 bytes means a short can hold 216 = 65536 distinct values. This is because a byte means 8 bits and a bit can be 2 values, so 28 values for 1 bye and 216 for 2.

* 1. void

The simplest type which means no value, it can’t be applied to variables but has other use cases.

Such as in functions, also in c

int abc(void) {

…

}

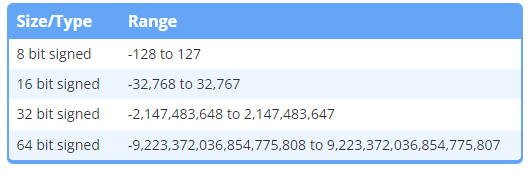
is valid. Which is why it is valid in C++ as well but is considered deprecated.

* 1. Signed Types

Due to how binary system works a type can choose to only hold positive values or sacrifice the Most Significant Bit to indicate the sign losing out on the larger possible range.

The signed types, which are the default types, do so and hence hold less

no. of distinct values but support -ve values.



int, long, double etc. are all signed types.

Basically

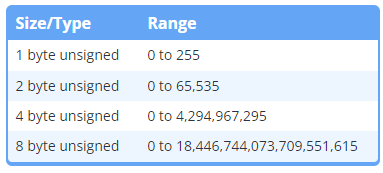
-(2n-1) to 2n-1-1

is the range of signed types where n represents the bits.

Ofcourse, only integral types have (un)signed types.

* 1. Unsigned Types

These types retain the MSB but now they can’t denote if the integer in binary is +ve or -ve.



As we can see, the range basically increases by 2x on the +ve end or more formally

0 to (2n)-1

To declare unsigned types we use the unsigned keyword

unsigned int x;

unsigned long long;

* 1. Integer Overflow

Since we know integral types have range of permitted values, any value outside ‘wraps around’, also called modulo wrapping.

For signed types, if a value exceeds the +ve limit it is stored on the -ve end with an increment of +ve limit - value.

For ex.:

if type T stores values in range -127 to 127 then a value of 128 would be stored as -127, a value of 129 would be stored as -126 and so on.

Similarly, if the value exceeds the -ve limit it is stored as the -ve limit + value.

So -127 to 127 for a value of -128 would wrap around to 127.

This is due to how binary system works.

For unsigned types the same happens but the values range between 0 to +ve limit.

For this reason it is inadvised to use unsigned types as when they are mixed with signed types, both the values are converted to unsigned types implicitly, this leads to loss of data and even undesired behavior sometimes.

For ex.:

int main()

{

signed int s { -1 };

unsigned int u { 1 };

if (s < u) // -1 is implicitly converted to 4294967295, and 4294967295 < 1 is false

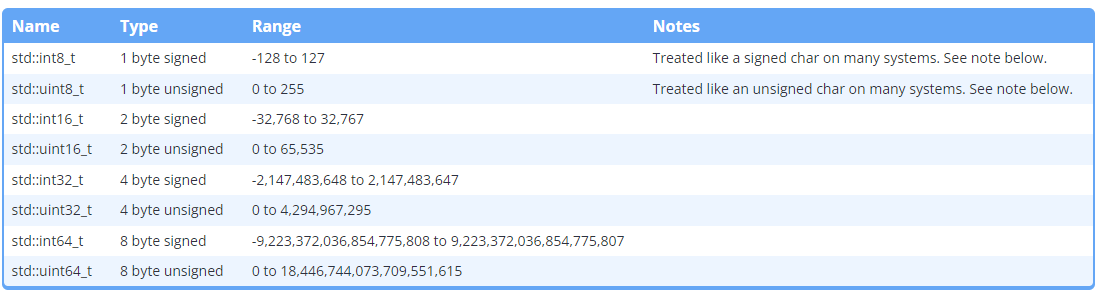
{…}

}

* 1. Fixed-width types

Depending on the host machine, the size of a type changes. That’s why the standard document only defines the ‘minimum size’ which is a guarantee that it will be atleast this big and has the max. compatibility.

However sometimes bigger sizes may be required, in that case C provides fixed-width types.



They are defined in the library

<cstdint>

The downside is that these types may not be defined for all platforms and even if they are they may not be optimal. For ex. if we use int32\_t then 4 bytes would be used but it could be that the host supports 8 byte ints, in which case the memory consumption would be greater by the program and possible lower performance too.

* 1. Fast and least ints

In the cstdint we also get fast and least types. These are there to resolve the issues with fixed-width types.

The std::int\_fast#\_t and std::uint\_fast#\_t provides types that are atleast given bits wide but are the fastest available types on the machine.

The std::int\_least#\_t and std::uint\_least#\_t provides types that are smallest size and atleast given bits wide.

For ex.:

sizeof(std::int\_fast16\_t) \* 8

could return 32 as it would mean that 32 bit types are faster on the given machine than 16 bit ones.

The downsides to these types is that if we rely on side effects of these types then undefined behavior may occur, such as -ve overflow on fast16\_t might produce different value on each machine.

* 1. 8 bit types

Due to an oversight in the standard documents, the 8 bit types (even fixed width, fast and least types) are treated as chars which can lead to bugs.

It is advised to avoid them.

* 1. size\_t

Just like the normal integers, size\_t is also dynamic but has a minimum size of 16bits and is unsigned by default. This is a special type because when operators like sizeof() return a value, it must be of some type, that type is size\_t.

size\_t is generally equivalent to the address-width of the application, i.e., on 32 bit machines it is 32 bits in size, and so on.

* 1. Other number systems

All numbers used in the source code in C++ are seen as numbers from the Decimal Number System. That is base 10.

But we can use other systems too, such as Octal and Hexadecimal. To do so we prefix the literal with 0 for Oct and 0x for Hex.

For ex.:

int x{012};

int y{0xF};

std::cout<<x<<” “<<y;

prints 10 and 15.

The cout converts the numbers back to the decimal system by default.

Binary Literals

From C++14 we can directly store binary numbers too (although all data types are stored as binary anyway, this allows for directly storing binary values). We use the 0b prefix.

For ex.:

int b{ 0b0101}; //stores 0101 in b, which means 5 in decimal system.

We can define cout to use a certain number system by using these I/O manipulators.

std::cout<<std::hex; //sets the cout to output all nums as hex

Similarly we have oct and dec.

* 1. Floating point

Numbers with fractional parts are represented by these 3 types, float, double and long double.

Since float and double use the same literals we use the suffix f to denote a number is of type float.

float x{2.0f};

double y{2.0};

C++ uses the scientific notation for large double values.

For ex.

std::cout << 9876543.21 << '\n';

prints 9.87654e+06

e means 10 and the digits to the right of it are the power of 10.

So here

9.87654e+06 means 9.87654\*107

The + means +ve exponent. The 0 here indicates padding.

Similarly, - would mean -ve exponent.

For ex.:

42030

would be denoted as

4.2030e4

in floating point.

and

0.0078900

as

7.8900e-3

The digits to the left of e are called significands and they denote the precision, the higher the precision the more the digits in the significand.

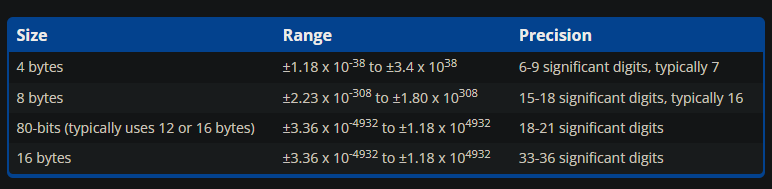
The reason the scientific notation doesn’t discard trailing 0s is to retain the information about precision.

In C++, the 0s after e are exactly that.

We can use the same notation to store FP values too

double electronCharge { 1.6e-19 }; // charge on an electron is 1.6 x 10^-19

* + 1. FP Range



* + 1. iomanip

In the example of

std::cout << 9876543.21 << '\n';

we can see that some digits are skipped by cout. This is because cout by default has a precision of 6 digits. We can change this precision by using an Output Manipulator, which is a function that alters how data is outputted by cout. In this case we use setprecision() function.

#include <iostream>

#include <iomanip> // for output manipulator std::setprecision()

int main()

{

std::cout << std::setprecision(16); // show 16 digits of precision

std::cout << 3.33333333333333333333333333333333333333f <<'\n'; // f suffix means float

std::cout << 3.33333333333333333333333333333333333333 << '\n'; // no suffix means double

return 0;

}

prints

3.333333253860474

3.333333333333334

That is also to say, despite the precision, if the type itself has a smaller precision then it will lose information. This is why it’s recommended to use double over float.

* + 1. Rounding Errors

Floating point in computers is not very intuitive, rounding errors are common because of how FP numbers are stored and moved around.

For ex.:

#include <iomanip> // for std::setprecision()

#include <iostream>

int main()

{

double d{0.1};

std::cout << d << '\n'; // use default cout precision of 6

std::cout << std::setprecision(17);

std::cout << d << '\n';

return 0;

}

prints

0.1

0.10000000…1

* + 1. nan and inf

FP in C++ (if the compiler supports IEEE 754 for floats) also supports infinity and non a number, which are special values.

#include <iostream>

int main()

{

double zero {0.0};

double posinf { 5.0 / zero }; // positive infinity

std::cout << posinf << '\n';

double neginf { -5.0 / zero }; // negative infinity

std::cout << neginf << '\n';

double nan { zero / zero }; // not a number (mathematically invalid)

std::cout << nan << '\n';

return 0;

}

prints

1.#INF

-1.#INF

1.#IND

* 1. Bools

Simple, true and false or 1 and 0.

bool a{true};

cout<<a; //prints 1

but if we use

std::cout << std::boolalpha; // print bools as true or false

cout<<a; //prints true

* + 1. Conversions

Integrals are implicitly converted to bools, 0 means false and all other ints mean true.

Strings are always true, chars are converted to their ascii values and then treated as ints.

* 1. Character

In computers, characters and all pieces of text are represented by integers. That is,

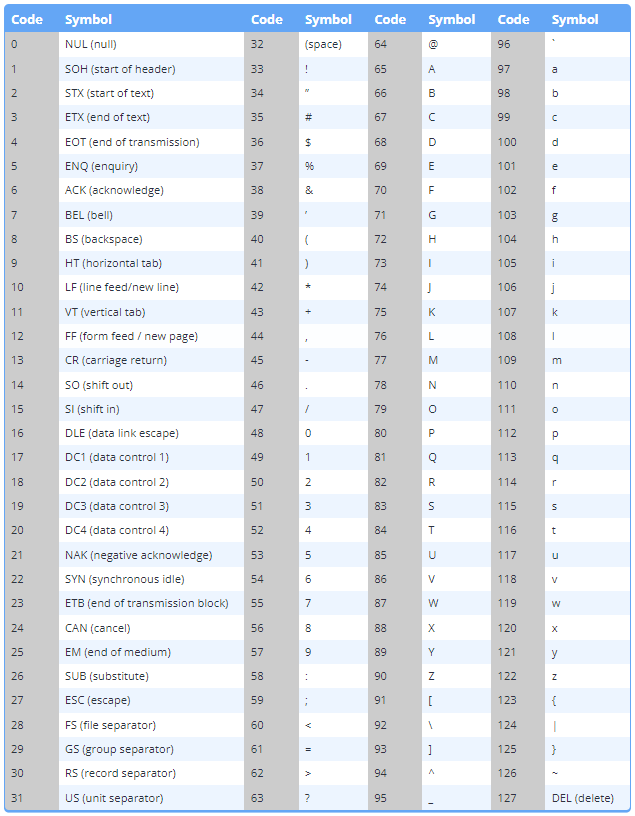
‘L’ has a unique integer mapped to it.

and same for other characters. These ints are unique per system classifying them. Whilst 42 is an int, a character x that is represented by this int may be anything, a letter such as ‘T’ or even ‘1’, the difference is that anything within single quotes is a character and is represented by an integer which may be different from its actual value.

C++’s default character type is char. Even strings in C++ are simply arrays of characters.

* + 1. Char

Follows the ASCII system for character mapping.



Stored like so

char x{‘a’}; //stores ‘a’ in x.

char x{97}; //same as above

They are the same thing because 97 actually maps to ‘a’. If we tried 1 and ‘1’, the value stored would be different since ‘1’ is represented by 49.

Char is 1 byte (28=256 values) and has a signed and an unsigned type, but we don’t need to define either because the range 0 to 127 is guaranteed either way. However we can still use signed char to store -ve ints.

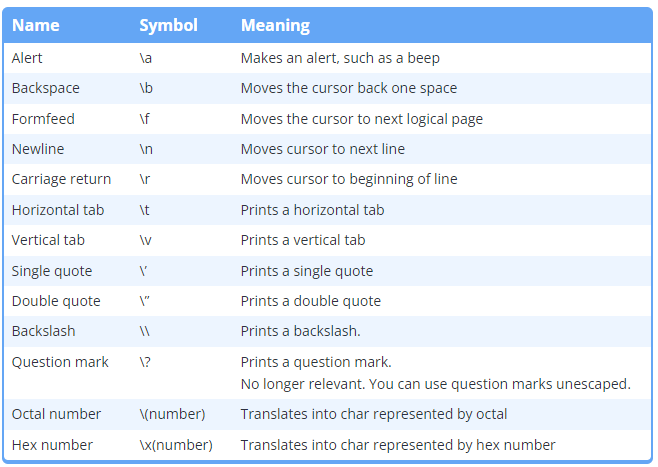
Escape Characters:

aka escape sequence, they convey special meaning to the compiler.

For ex.:

‘\n’ means add a newline character to the string.

Note that these are still seen as single byte characters so they must be within single quotes.



C++, for backwards compatibility, still supports multi-character literals but its inadvised to rely on them as they aren’t part of the standard and are hence compiler dependent.

That is,

‘/n’ (not \n) has 2 characters within the same single quotes so it will cause undefined behavior.

Chars can be part of string but if a single char is to be used, prefer single quotes to allow compiler better optimization.

std::cout<<”yo \n” <<”\n”; //is okay but prefer

std::cout<<”yo \n” <<’\n’;

The 8 bit integral types in C++ may be seen as char types by some compilers, this is why they must be avoided.

* + 1. Unicode

The default char only supports 127 english letters but a lot more letters exist in the world, they are represented by Unicode which is a 32-bit system so 232 possible letters are supported, called UTF-32. Smaller unicodes also exist such called UTF-16 and UTF-8.

In C++ we can use these using the types

char16\_t for UTF-16

and char32\_t for UTF-32

There’s also wchar\_t but it is deprecated and it’s size is implementation defined.

* 1. Bitset

A simple compile time fixed size data type to represent binary numbers,

For ex.:

std::bitset<4> c{1100};

std::cout<<c; //prints 1100

4 of its methods are pretty useful,

set(<pos>); //sets the pos of the bitset as 1

reset(<pos>); //sets pos as 0

flip(<pos>)

test(<pos>); //returns the bit

The size of a bitset is rounded up to sizeof(size\_t), so for 32 bit machines a bitset will be of size 4 bytes if < 32 bits and for 64 bit it will be of 8 bytes if < 64 bits.

For ex.:

std::bitset<1> will take size 4 bytes on 32 bit machines.

This is to optimize for speed as it means 4 byte values are fastest on that machine.

* + 1. Bitmasks

The usual ANDing or ORing of bitset to modify a bitset is quite simple in C++.

constexpr std::uint8\_t someFlag{ 1 << 1 };

std::bitset<8> yo {};

yo |= someFlag;

if (yo && someFlag){  
 …

}

works.

This is useful to store a lot of Boolean states in just a single variable.

* 1. String

String literals such as “uwu” are called C-style strings because they are stored as character arrays rather than as C++ string data type.

The C++ string data type is defined in the <string> library.

For ex.:

std::string yo{“lool”};

and they work as expected, directly prints out using cout and can be assigned normal C-style strings (which are converted into string data type).

* + 1. I/O

We can’t directly use cin to get strings in C++, this is because of how cin works.

By default, cin uses the input buffer. It uses its default mode called “extraction mode” where it reads all valid characters and passes them into the variables whilst ignoring whitespaces and newlines.  
But when the target variable is of type string, even whitespaces could be a part of the string so it wouldn’t know when to stop accepting characters, to avoid this situation it simply stops on first newline or whitespace, avoiding leading whitespaces or newlines.

For ex.:

std::string yo{};

std::cin>>yo;

std::cout<<yo<<’\n’;

std::cin>>yo;

std::cout<<yo<<’\n’;

When we input “Hewwo Wowld”, only “Hewwo” is in yo and not the “ Wowld” and in the next cin, this “Wowld” is directly picked up without waiting for user input as it was in the input buffer and we get

“Hewwo”

“Wowld”

as output which is likely not expected behavior.

To circumvent this we use 2 things, std::getline and std::ws Input Manipulator.

So,

std::string yo{};

std::getline(std::cin>>std::ws,yo);

Getline is a function that takes an input stream, such as cin and reads it until it finds a newline character, then it puts all the read content into the given variable.

std::ws is an input manipulator that sets cin to ignore any leading whitespaces or newlines since it’ll affect getline if there is any in the stream already.

For ex.:

int x{};

std::cin>>x;

std::string yo{};

std::getline(std::cin,yo);

If we enter “1” then “yo” then it will pick up 1 and then leave the ‘\n’ at the end of its line in the stream, then when getline uses the stream it will find “\n” in it and immediately exit leaving yo as empty.

* + 1. Size

Use <String variable>.length() to get the no. of characters in it. From C++20, we can also use std::ssize(<string variable>).

* + 1. String Literal

Use the suffix s after a literal to directly get it as string type.

* + 1. Constant String

Not supported yet.

That is,

constexpr std::string yo{“yoo”s}; //error

right now.

* + 1. Raw String  
       More read: <https://stackoverflow.com/a/56710099/13036358>   
       and <https://en.cppreference.com/w/cpp/language/string_literal>

Use R”(<str>)” for raw string. Accepts delimiters too.

* + 1. String\_view

C++17 introduces these as they are faster than string data type to copy and pass around. Defined in #include<string\_view>.

To use them,

std::string\_view s{ "Hello, world!" };

And they should be passed around to functions as such. They are read-only so characters can’t be modified, and they actually only provide access to the underlying string without copying it.

constexpr std::string\_view s{ "Hello, world!" };

is fully supported and advised to be used.

string implicitly converts to string view but we need to explicitly convert for other way around. string can be initialized from string\_view without explicit conversion though.

To convert string to C-Style string

std::string str {…}

auto cstring{ str.c\_str()};

sv can’t use this method, it can be converted to a string though, which can convert to C-Style String.

sv can use an unrecommended method, data() to get a C-style string, it technically returns the pointer to the first character referenced by the string\_view. it is unrecommended because unlike c\_str(), string\_view’s data() doesn’t null-terminate strings and if they are already non-null terminated, it can lead to undefined behavior.

For ex.:

#include<string\_view>

#include<iostream>

#include<cstring>

int main()

{

std::string\_view s{“Hallo”}; //char\* Hallo is null-terminated as it is a char\*.

std::cout<<s.data();//prints “Hallo”, the null character exists

s.remove\_prefix(1);

s.remove\_suffix(1);

std::cout<<std::strlen(s)<<” “<<std::strlen(s.data()); //prints 3 4

std::cout<<s.data(); //prints “allo”

std::cout<<s; //prints “all”

}

data doesn’t check for null character, but it returns pointer to first character the sv is referencing, so “a” in this case, then strlen tries to find null char on C-Style String, which exceeds crosses into adjacent memory, same with cout.

Literal suffix: sv

Useful methods that apply to both string and string\_view

They are faster on string\_view as they only return sv which is not a copy but a kind of reference of a string.

std::string\_view s {“yoo I am pink”};

s.substr(<start index>,<end index>); //returns an sv in given range

s.find(<char or string>); //returns the index where given char or string starts

s.starts\_with(“<string>”);// C++20

s.ends\_with(“<…>”); //C++20

SV are built on top of a C-Style String or String, meaning they provide an immutable reference over the string.

For ex.:

#include<string>

#include<string\_view>

#include<iostream>

int main(){

std::string s{“yoo”};

char c[]{“hallo”};

std::string\_view v{s}; //ok, v can’t be modified and doesn’t copy s, just provides a reference to it.

v=c; //also works

c[2]= ‘o’;

std::cout<<v; //prints “haolo”

}

And just like a reference if the main value is changed, the reference/sv is updated too.

The only thing to be wary of with them is to not return string view from functions unless the duration can be guaranteed . As they are always a reference, even std::string\_view v{“yoo”}; is a reference over a string “yoo”, so when we pass v around, if the string “yoo”’s scope is over, it is destroyed and string\_view is referencing an invalid string.

That is, treat string\_view exactly like a reference.

Shrinking

String\_view is like a reference, but it can decide what length of a string it wants to show, we can shrink that referenced string with remove\_suffix and remove\_prefix.

For ex.:

std::string\_view v{“hallo”};

v.remove\_prefix(1); //removes first char from sv

v.remove\_suffix(2); //removes last 2 chars from sv

This is not modifying the original string, only the part that is visible in the reference.

The size can’t be widened, only reassignment will bring the original string’s reference back.

std::string and std::string\_view both can be initialized or assigned with non-null terminated C-Style Strings, because unlike C-Style strings, these strings use length as a property to keep track of used characters.

* 1. Compound Data Types

Data types that are constructed from fundamental types. Such as arrays.

C++ has these CDTs

* Functions
* Arrays
* Pointers: Pointer to Object, Pointer to Function
* Pointer to Member Types: Pointer to Data member, Pointer to Member Function
* Reference Types: L-Value Refs, R-Value Refs.
* Enumerated Types: Unscoped enums, Scoped Enums
* Class Types: Structs, Classes, Unions
  + 1. Entity and Identity

An entity is any object or function or the like.

It’s identity is basically the name, type and other details.

Entities with identity can be compared to other entities by simply comparing the value at their addresses using an identifier (like name) or references or pointers and have longer lifetimes than single statements and exprs.

That is,

int x{2};

int y{3};

2+2;

Here x and y are the identifiers of variables with type int. We can compare x and y by comparing the values contained within (x==y) or by comparing their references or pointers.

* + 1. L and R Value

Every expression in C++ has 2 properties, a value and a type. C++ requires the type of an expr. to be deducible at compile time, but the value can be defined at compile time (constexpr) or runtime.

For ex.:

auto x{2/2};

Here x has the type int, if the expr. 2/2 didn’t give a type to initialize x then it would have been an error.

L-Value: An expr. that evaluates to an identifiable object or function or bit-field.

For ex.:

int x{2};

int y{x};

Here x in line 2 is an L-value because the expr. evaluates to an entity with identity, that is the variable x.

There are 2 types of L-Values, modifiable (such as variable y above) and non-modifiable ones (such as constexpr variables).

R-Value: An expr. that evaluates to a non-identifiable temporary value.

For ex.:

int x{};

x=5;

int y{someFunction()};

int z{x+1};

All of these are R-Value as they evaluate to a temporary value.

Assignment operation requires the left operand to be a modifiable L-value and the right operand to be an R-Value.

So,

x=5; //works

5=x; //doesn’t

And L-Values implicitly convert into R-Values so

x=y; //also works

PR-Value, GL-Value, X-Value in modern C++ are further subdivisions of R-Values.

* + 1. References

A reference is like an alias for an existing object. The use for these things is to allow operations on objects without copying them around in the memory and using the same address everywhere.

2 types of refs in C++, L-Value Ref and R-Value Ref.

* + - * 1. L-Value Ref

A reference to an existing L-value.

For ex.:

int x{}; //L-value

int& y {x}; //L-Value Ref

int& is an L-Value reference type (and not address of x),

we use the variable y just like we would use x, the benefit is that all operations on y do the same to x, and also vice versa.

LVRs must be initialized and bound to an L-Value, when an LVR refers to another entity, it is said to be bound, the entity is called the referent and this process is called reference binding.

For ex.:

int& x; //error

int& y{}; //error

int& k{2}; //error

int z{};

int& v{z}; //works

const int m{};

int& n{m}; //error, can’t have a normal LVR to a const L-Value

This is why LVRs are aka non-const references.

In most cases the LVR must be bound to an L-value of the same type.

LVRs can’t be reseated, that is, once initialized and bound to an L-value, they can’t be bound to another L-Value.

For ex.:

int x{0};

int y{2};

int& ref{x};

ref=y;

std::cout<<x<<’\n’; //prints 2

ref doesn’t refer to y here, y is evaluated and converted into an R-Value and that R-value is copied to ref (and x).

LVRs have the same scope and duration as normal variables and their lifetime isn’t bound with the referent either.

So,

int x{2};

{

int& y{x};

y=4;

}

std::cout<<x; //prints 4

is valid.

However if the referent dies before the reference then that isn’t an error but it leads to undefined behavior and this reference is called a dangling reference.

Unlike Rust, a single referent can have multiple references.

LVR can’t be bound to another LVR.

For ex.:

int x{};

int& y{x};

int& z{y};

both y and z refer to x and z doesn’t refer to a reference, the reference y is evaluated to x and z takes that instead. That is, Reference to a Reference isn’t supported in C++.

LVRs are not objects. And unlike objects, it is not required to take up storage and exist. The compiler **may** optimize away the references at compile time itself and replace them with the referents, and they are indeed aliases. This is why LVRs are not called reference variables as variables require storage.

However, in some cases LVRs may need storage.  
  
In Loops, LVRs will not be optimized away.  
For ex.:

int[] a{1, 2, 3};

for(int i{}; i < 3; ++i)

{

int& b{a[i]};

}

Here, the compiler doesn’t optimize away the reference b. Instead, it creates a new reference/alias to a[i] in b for each value of a. So whatever operation we perform on b will occur on a[i], all without using any pointers or raw addresses, just an LVR which is simply an alias to a[i].

Const LVRs:

An LVR can’t reference a const value because modifying it through the LVR wouldn’t make sense. But we can make LVRs constant with the const keyword, these are called const references or LVR to const.

For ex.:

const int x{2};

const int& y{x};

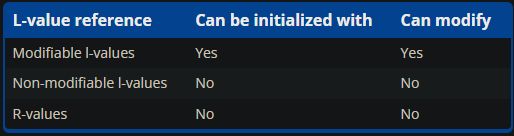
works. We can access the value of x through y but can’t modify.

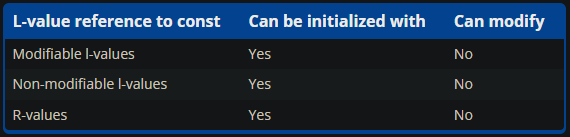
const refs can also be bound to non-const L-Values and also R-Values.

const int& z{5}; //works

5 is a temporary/anonymous object as it is ephemeral by itself and has no identifier but using const ref increases its lifetime to match that of the const ref as this is a special rule of C++.

This table summarises the different types of LVRs





* + - 1. Reference to Pointer

It is a thing in C++ and allowed.

For ex.:

int x{};

int\* p{&x};

int\*& ref{p}; //works

int y{2};

ref= &y; //assigns y’s memory address to p.

std::cout<< \*ref; //print ref’s referent’s memory address’ value, so print 2

* + - 1. Check for reference

typeid returns the type info but not the reference status for an object, so typeid of a reference is equivalent to the typeid of a value type.

We use std::is\_reference type trait to check if an object is a reference, is\_lvalue\_reference to check if an object is a reference to an lvalue and is\_const with remove\_reference to check if the object is const

For ex.:

#include<type\_traits>

int main() {

int x{2};

int& ref{x};

std::cout<<std::boolalpha;

std::cout<< std::is\_reference<decltype(ref)>::value<<’\n’; //prints true

std::cout<< std::is\_const<std::remove\_reference\_t<decltype(ref)>>::value<<’\n’; //is const ? false

std::cout<< std::is\_lvalue\_reference<decltype(ref)>::value<<’\n’; //prints true

}

LVR and RVR can be differentiated using is\_lvalue\_reference.

* + - 1. R-Value References

RVRs are introduced by C++11 and can be only initialized with R-Values. We use double & to denote an RVR.

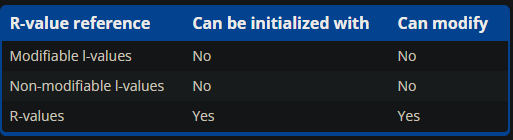
For ex.:

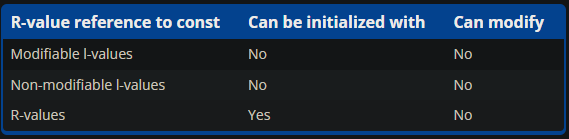
int x{2}; //L-Value

int& lx{x}; //LVR

int&& rx{5}; //RVR

This table summarises different types of RVRs





When an RVR is created, it extends the lifetime of the temporary object in it to match that of the identifier.

The type of reference does matter in function overloading.

For ex.:

#include<iostream>

void yo(int& x) {

std::cout<<”LVR”<<’\n’;

}

void yo(int&& x) {

std::cout<<”RVR”<<’\n’;

}

int main() {

int x{2};

yo(x); //prints LVR

yo(3); //prints RVR

int&& rx{3};

yo(rx); //also prints LVR

}

The last statement prints LVR because RVRs are L-Values too. This is because the value category of a variable is independent of its type, so whilst the type of rx is R-Value Reference, its value category is an L-Value just like any normal variable.

Just like LVRs, RVRs mustn’t be returned if the lifetime of the RVR doesn’t exceed that of the function, in that case the reference will be a dangling/hanging reference as the temporary object will be cleared.

* + 1. Pointers

Every entity with identity in C++ occupies some space in the RAM. This space is denoted by its address in the memory and C++ abstracts away the address from the types so we work directly with the values stored at that address.

We can get the address of an entity with &<object>.

For ex.:

int x{};

std::cout<<x; //prints 0

std::cout<<&x; //prints the memory address of x.

The value of x is stored at that address and we can manually get the value at that address (using something like Cheat Engine or Process Hacker).

We can get the value at an address using the dereference operator \*, so

std::cout<< \*(&x); //prints the value of x

It returns the value as an L-Value.

A raw pointer is a data type in C++ that stores the memory address of an object.

For ex.:

int\* x; //x is of type int pointer

int\* y, z; //y is int\* but z is just int

int \*m, \*n; //both are pointers

Uninitialized pointers hold garbage address, it is recommended to initialize them,

int\* x{}; //works, and x is a null pointer

int x1{2};

int\* y{&x1}; //to initialize with x1’s address

std::cout<< \*y; //prints the value held by the address in pointer.

A raw pointer can only point to a value of same type and initializing pointers with R-Values is not allowed in C++ either.

To change the pointed address

int x{};

int y{};

int\* p;

\*p = 6; //dereferences p and puts 6 at the memory address of p

p = &y; //assigns p the address of y

std::cout<< \*p; //prints 0

Pointers are very similar to references in how they work, but they are quite different in implementation. Ptrs take up memory, can be reseated, don’t need to be initialized, aren’t safe, can point to nothing as well and are objects.

Pointers point to objects, and references aren’t objects so pointer to reference is not allowed.

Now, since pointers hold addresses and addresses are retrieved by &<object>, here, we get

int x{};

int\* p{&x};

std::cout<< p << “ ” <<typeof(&x).name();

prints “<address of x> int \*”, can also print “… pi” where pi means pointer to int.

That is, & operator actually returns a pointer type, and a pointer type is nothing but a storage for memory address.

* + - 1. Size

The size of any pointer, regardless of its type, is fixed at either 4 bytes for 32-bit machines or 8 bytes for 64-bit machines.

That is to say, a pointer is a memory address in integer and takes 4 bytes or 8 bytes to store any other entity’s address in it which is also a memory address in integer.

* + - 1. Dangling Pointer

If the object is destroyed at the memory address or pointer and the pointer still tries to access it, it will be undefined behavior, such pointers are called Dangling Pointers.

For ex.:

int\* p;

{ int x{};

p= &x;

}

std::cout<< \*p; //works but the behavior is undefined.

* + - 1. Null Pointer

Raw Pointers canhold 2 types of value, a memory address or a null value. Whilst C++ doesn’t use null anywhere and no type accepts it, pointers are the exception. We use nullptr value to set a pointer’s value as null.

To get a null pointer we can do

int\* x{}; //x is initialized as a null pointer

int\* x2{nullptr}; //same

int\* x3; //unitialized pointer, points to a garbage memory address

x3= nullptr; //set x3 to nullptr, null value

int\* y1{NULL}; //null definition from C, defined in #include <cstddef>

int\* y2{0}; //implementation specific

nullptr is the modern way of doing it. 0 and NULL are not standardized.

Boolean

If a pointer has a null value, it returns false when converted to Boolean.

int\* x{};

if(x) {…} //x is false, read as “if x is not null”

if(x==nullptr); //true

Dangling pointers aren’t implicitly made null pointers so it is the responsibility of the programmer to set the pointer to null when the object it is pointing to has been destroyed.

* + - 1. Raw Pointers to Const Object

Normal raw pointers can’t point to const objects, so

const int x{};

int\* p{&x};

is an error as the type of &x is const int \* but p is of type int \*.

Just like const references, we can define pointers to const objects

const int x{};

const int\* p{&x}; //works and now

\*p = 2; //is an error

Raw Pointers To Const Object, RPCO can point to non-const objects too.

There’s also const pointers, these are pointers that theirselves can’t point to another object after initialization

int x{};

int\* const p{&x};

int y{};

p= &y; //error

\*p= 6; //ok

The value at the address can still be changed.

Const Pointer to Const Object

Lastly, we can also have const pointers to const objects

const int x{};

const int\* const p{&x};

now neither the value nor the memory address of p can be changed.

* + - 1. std::nullptr\_t

nullptr is an actual value, although when compared to an int, it is promoted to an integer, which is 0. So nullptr==0 is true but that is due to numeric promotion.

The type of nullptr is nullptr\_t (defined in cstddef)

For ex.:

#include<iostream>

#include<cstddef>

void print(std::nullptr\_t) {

std::cout<<”nullptr”;

}

void print(int\*) {

std::cout<<”int\*”;

}

int main() {

print(nullptr); //prints nullptr

int x{};

int\* p{&x};

print(p); //prints int\*

p=nullptr;

print(p); //prints int\* since type of p is still int\*

print(\*p); //error since type of \*p is int and there is no overload for int.

}

* + - 1. Arithmetic

Say p is a pointer, then p+1 returns the next object’s memory address and p-1 returns the previous object’s memory address.

This is not the same as the next memory address, say p is an int\* then the next object starts after 4 bytes so ptr+1 means memory address after 4 bytes. Similarly, ptr+2 means memory address after 2 of the same object, so memory address after 8 bytes and so on.

In pointer arithmetic, the integer operand is multiplied by the size of the object the pointer is of, this is called scaling.

For ex.:

#include<iostream>

int main() {

int v{2};

int\* p {&v};

std::cout<< p; //prints 0x1234

std::cout<< p+1; //prints 0x1238

std::cout<<p+2; //prints 0x1242

}

* + - 1. Void Pointer

aka generic pointer, this pointer can point to an object of any type.

#include<iostream>

#include<string>

int main() {

int a{2};

double s{2.0};

void\* p {&a}; //ok

p=nullptr; //ok

p= &s; //ok

std::cout<< \*p; //error

std::cout<< \*(static\_cast<double\*>(p)); //ok

}

Whilst it can hold a memory address of any type, it must be casted into the proper type’s pointer before being dereferenced.

There’s no such thing as void reference and pointer arithmetic doesn’t work on void\*.

* + - 1. Pointer to pointer

The type in T\*\*.

For ex.:

#include<iostream>

int main() {

int a{2};

int\* p{&a};

int\*\* pp{&p};

std::cout<< \*pp; //prints the address of p

std::cout<< \*\*pp; //prints 2

int\*\* pp2 {&&a}; //error

}

The reason &&a is an error is because & operator requires an lvalue but &a is an rvalue, alternatively or additionally, to have a pointer to a pointer, we must first store the first pointer in memory.

Still, int\*\* can take nullptr.

* + 1. Program-Defined Types

Commonly mistakenly known as User-defined types. These are custom types defined for each program.

For ex.:

struct Fraction {

int num{};

int denominator{};

}; //notice the semicolon, it is required

int main() {

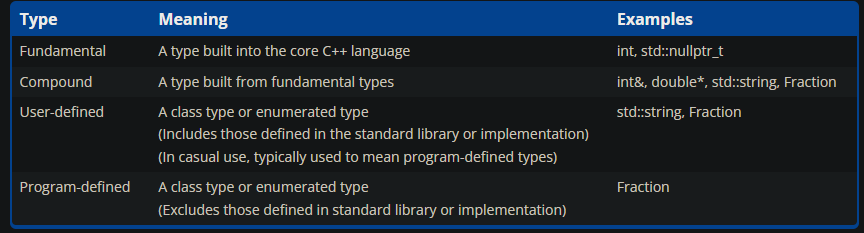
Fraction fraction{3,4};

}

works, here struct is used to give a definition for a type named Fraction, it isn’t an object by itself. But when we make an object fraction of type Fraction, then that fraction itself is an object.

PDTs can not be forward declared, the compiler needs the definition. However, PDTs are partially exempt from ODR, that is, they can have multiple definitions, but if in a code file there are multiple definitions and they aren’t identical then that leads to an undefined behavior.

PDTs are a subset of User-defined types because UDTs, as defined by the standard, are types that are defined by the user, or the library, or the standard library and PDTs are only the types that are defined by user.



* + 1. Aggregate Data Types

Any data type that can contain multiple data members, like struct, array etc. Some ADTs allow multiple data types like structs or don’t, like arrays.

To be an aggregate, these conditions must be met

* Is a class type (a struct, class, or union), or an array type (a built-in array or std::array).
* Has no private or protected non-static data members
* Has no user-declared or inherited constructors
* Has no base classes
* Has no virtual member functions

ADTs are special because they have certain behavior defined for them, like aggregate initializer (being initialized with an initializer list {…}) etc.

* + 1. Enumeration

It’s a compound type where every possible value has to be defined as a symbolic constant, that is a constant value with a symbol, aka an enumerator.

We can define these too so they are PDTs as well.

For ex.:

enum Color {

kRed, //This symbolic constant is a value of enumerated type Color, i.e., an enumerator.

kGreen,

kBlue,

}; //semicolon necessary for PDT

int main() {

Color a {kRed};

Color b {kBlue};

if(a==b) {…} //works, false here

}

It is a naming convention to use k<PascalCase> with enmurators.

An enumerated type doesn’t need to be named but it is recommended to have named enumerations only.

Enumerators are distinct from each other and enumerated types are also distinct from each other. So 2 same enumerators for 2 distinct enumerated type are not the same type. However the unscoped enumerators are still comparable due to being implicitly integral.

For ex.:

enum color {…}

enum shape {…}

int main() {

color c{…};

shape s{…};

if(c==s) {…} //works and is not an error

}

An enumerator is a value in the scope it is defined in (and deeper ones). Enumerated types defined in the global scope pollute the global scope with their enumerators, and are called unscoped enumerators.

Unscoped enumerations can also be defined in custom, for ex. in a namespace or class or even a function.

For ex.:

namespace color {

enum Color {

kRed

};

}

int main() {

color::Color c {color::kRed};

enum wow {…

}; //works and is usable in this scope from this point on

}

* + - 1. Integral value

Every symbolic constant in an enumeration has an integer associated with it.

For ex.:

enum Color {

kRed, //auto assigned 0

kBlue, //1

kGreen = 5, //explicitly assign 5

kYellow, //auto assigns next unused int, so 6

};

int main() {

Color red{ kRed}; //actually stores 0

}

enumerators get ints assigned to them implicitly, by default any enumerator gets last enumerator’s int value + 1, and they start at 0, but we can change that by explicitly assigning an int value to any enumerator.

Each enumerated type has a base, which is the integral type that represents its values. By default it is int, but we can explicitly change it too

For ex.:

enum color: uint16\_t {

…

}

will have all enumerators of color be an integral value of type uint16\_t.

enumerator from int

To assign an enumerator using an int value into an object,

enum color {…};

enum color2 : int

{…};

int main() {

color a{2}; //error

color b{static\_cast<color>(2)}; //ok

b= static\_cast<color>(3); //also ok

color2 c{2}; //ok, from C++17

c=3; //still an error

}

We can use operator overloading on >> and << to allow cin and cout to directly work with enumerators, otherwise we can simply cast to int both ways to do the same without operator overloading.

Unscoped enumerators get numeric promotion to become ints and hence are comparable even whilst belonging to different enumerations,

for ex.:

enum color {…}

enum shape {…}

int main() {

color c{…};

shape s{…};

if(c==s) {…} //works and is not an error

}

Here c and s are promoted into their integral values and hence get compared, and can be equal despite belonging to different enumerations.

All other properties apply to scoped enumerations too.

* + - 1. Scoped Enumeration

These enumerations are strongly typed and strongly scoped. Meaning they don’t implicitly convert to ints and have their values in their own scope. Defined using enum class.

For ex.:

enum class Color {

red,

blue

};

enum class Dog {

NotDog,

HotDog

};

int main() {

Color c {Color::red}; //scoped

Dog d {Dog::NotDog};

if (c==d) {…} //error

}

The enumerators are scoped so they have to be prefixed with the scope.

Using enum, introduced in C++20, works much like using namespace excepts it opens the scope of the enumeration.

For ex.:

enum class A {…}

int main() {

using enum A;

A a{kA}; //works without the scope op.

}

* + - 1. Enumerator to Int Trick

#include <iostream>

enum class Animals

{

chicken, // 0

dog, // 1

cat, // 2

elephant, // 3

duck, // 4

snake, // 5

maxAnimals,

};

 // adapted from https://stackoverflow.com/a/42198760

constexpr auto operator+(Animals a) noexcept

{

return static\_cast<std::underlying\_type\_t<Animals>>(a);

}

int main()

{

std::cout << +Animals::elephant << '\n'; //prints 3

return 0;

}

By overloading the unary + we can access an enumeration’s int value (much like a dereference operator).

* + 1. Structs

Another type of PDT. These allow us to bundle multiple types into a single type.

For ex.:

struct Animal {

int age{}; //A data member/ member variable.

std::string name{};

}; //semicolon needed for PDT definitions

int main() {

Animal cow;

cow.age=24;

std::cout<<cow.name;

}

The definition itself is not an object, but the object of the struct is an object.

A member of any struct is a variable/function/type defined within it. We can access these wth the member selection operator ‘ . ‘.

Since we defined the members above with default value, we don’t need to initialize them when an object is created. But members can be defined without an initializer,

For ex.:

struct Animal {

int age;

std::string name;

};

int main() {

Animal cow;

cow.age=24; //ok, and now it is initialized

std::cout<<cow.name; //undefined behavior, name is not initialized

}

When members are given default values in definition, it’s called non-static member initialization and the values theirselves are called default member initializers.

* + - 1. Aggregate Initializer

Using this we can initialize all members of a struct at once.

For ex.:

struct Animal {

int age;

std::string name;

};

int main() {

Animal cow { 1, “yo”}; //list initialization

Animal cow2 = {2, “aa”}; //copy-list init.

Animal cow3 (3, “noo”); //direct init. from C++20

}

Recommended to use list initialization.

If some members are left in Aggregate Initializer, they are initialized to default values of their type.

Meaning,

Animal cow { }; //initializes all members to default vals

Aggregate initializers can also be used for assignment.

For ex.:

Animal cow {1, “yo”};

cow = {2}; //leaves name as-is.

Designated Initializers

From C++20, we can initialize struct members by their names.

For ex.:

struct X{

int a;

int b;

int c;

};

int main() {

X x {.a { 1 }, .c{ 2 } }; //ok, b is auto initialized to 0

X y {.c { 2 }, .a { 1 } }; //error

X z {};

z= {.a {1}}; //ok

}

It’s an error in the 2nd initializer because the order still matters, and the members getting initialized must follow the order of declaration.

Explicit initialization takes precedence over default member initializer (if defined).

* + - 1. const struct

const <struct> <objectname> to make a struct const. This is a low-level const so member values can’t change for const structs.

* + - 1. Nested Structs

We can do it in 2 ways,

struct A {…}

struct B {

A a {};

}

//or

struct C {

struct D { //can be accessed with C::D

int x;

}

int i;

D d{};

}

int main() {

B b { {…} };

C c {2, { 3 } };

C::D d {2}; //works

}

If we nest the definition itself, then the nested type can only be accessed through the parent struct’s scope.

* + - 1. Struct Size

For ex.:

#include<iostream>

struct Cow{

short a;

int b;

Double d;

}

int main() {

std::cout<<sizeof(Cow);

}

should print 2+4+8=14 but it could print 16, where the extra 2 bytes are empty space not reserved for anything. This is due to padding, which is an optimization the compiler makes for performance reasons.

The term here is Alignment in memory. And this is very sensitive, the size of a struct can change drastically just based on which type is defined first.

* + - 1. Pointer and References to Structs

Works as normal but the member selection operator needs to account for the type.

That is,

struct X{

int i;

};

struct Z {

X x;

}

int main() {

X x {2};

X\* y{&x}; //get x’s memory address/pointer into y

std::cout<< y.i; //error

std::cout<< (\*y).i; //ok

std::cout << y->i; //ok, called member selection to pointer operator

Z z { x };

Z\* zp {&z};

std::cout<< zp->x.i ; //works, this expr. is evaluated LTR

std::cout<< (zp->x).i; //works, better readability

}

For a class/struct type T\*, we can use member selection operation -> to access members inside the object. This pointer is called a pointer to members.

* + - 1. Passing and Returning Structs from functions

Similar to normal data types, passing by value, reference or pointers work. Same for returning.

Some optimizations can still be made. For ex.:

struct yo{…};

yo getYo() {

return yo {…}; //use this instead of creating a new variable and returning that.

};

This avoids extra copy.

We can also use return { };

This means value initialize all members to 0 then return the struct of type given by function’s return type.

* + - 1. Class Template

Templates to make type definitions generic, these can be applied to classes, structs and unions.

For ex.:

template<typename T>

struct X{

T a; //template type

T b;

int c; //non-template type, always int

};

template<typename T, typename U>

struct Y {

T a;

U b;

}

int main() {

X<int> x{1,2, 3}; //instantiates an integer definition of X, then initializes an object x

X<int> y{2,4, 3}; //uses instantiated int definition

X<double> z{2,4.0, 3}; //instantiates a double definition, then initializes z by casting //2(int) to double

Y<int,double> k {2,4.0}; //ok

}

Just like Function Templates, the template parameter declaration defines the template types which we can use in the struct definition, then when a specific type of struct with T as any type, it’s definition is created if it doesn’t already exist or uses the already created definition to initialize the object.

And just like Function Templates, we can explicitly initialize a struct with type T

For ex.:

template<typename T>

struct X; //ok, as long as this forward declaration has definitions for T types that are used

template<>

struct X<int> {

…

};

template<>

struct X<double> {

…

};

int main() {

X<int> x{…}; //works

X<double> y{…}; //works

X<char> z {…}; //error, X has no definition for char type.

}

Class Templates with functions

template<typename T>

struct X{…};

template<typename T>

T yo(X<T> x) {…}

int main() {

X<int> x{};

yo<int>(x); //ok

yo(x); //deduces the T for yo from X’s T

}

For function overloading, X<int> and X<double> as params are treated as different params so it works. Explicit Specialization works for function templates using class templates as well, just replace T with types.

For ex.:

template<typename T, int sz>

struct X {

T a;

};

void pp(X<int,3> x) { }

int main() {

X<int, 3> x{2};

pp(x); //ok

}

Partial Template Specialization

We can also specialize pp() for only a certain template params, for ex.:

template<typename T, int sz>

struct X {

T a;

};

template<int sz>

void pp(X<int,sz> x) { }

int main() {

X<int, 3> x{2};

pp(x); //ok

}

This matches all pp() calls when X is int with any int size.

The std library has a type std::pair<T,U> already that works much like a class template struct with 2 values. It is in #include<utility>.

As we have seen, class templates are ok with forward declaration and redefinition, so much like function templates, they can be forward declared in source files and defined in headers.

* + - 1. Class Template Argument Deduction

From C++17, we can let the compiler deduce the types for class templates.

For ex.:

#include<utility>

int main() {

std::pair<int,double> p1 {1,2.0}; //ok

std::pair p2 {1,2.0}; //ok with CTAD, T is int, U is double

std::pair<int> p3 {1,2.0}; //error, partial type definitions are not CTADed

std::pair<> p4 {1,2.0}; //same error

}

However, it doesn’t work for aggregate initializers for our own PDTs with multiple template types. To assist with this we define a ‘deduction guide’, this special statement allows CTAD to deduce the types for our class templates.

For ex.:

template<typename T, typename U>

struct X{

T a;

U b;

};

//deduction guide

template<typename T, typename U>

X(T,U) -> X<T,U>;

int main() {

X x {1,2.0}; //ok

}

Without the deduction guide it would fail to compile as CTAD can’t deduce class template type when given aggregate initializer for our own types.

The deduction guide takes a template parameter definition and our type name, then we indicate what template types would our type be called with, in this case T and U, on the left side and on the right side we indicate what those template types map to the template types of our type.

Types such as std::pair come with pre-defined deduction guides so they don’t need these.

We can also do some odd things with deduction guides, for ex.:

template<typename T >

struct X{

T a;

T b;

};

//deduction guide

template<typename T, typename U>

X(T,T) -> X<T>;

int main() {

X x {1,2}; //works, deduced as <int,int> then mapped to X<int>

}

C++20 can automatically generate deduction guides.

Just like Function Templates, class templates can also define default template types

For ex.:

template<typename T=int, typename U=int>

struct X{

T a;

U b;

};

//deduction guide

template<typename T, typename U>

X(T,U) -> X<T,U>;

int main() {

X x {1,2.0}; //ok

X y; //ok

X<int> z{…}; //ok

}

* + - 1. Class Template in Header

Using forward declaration of methods with class template is not as straightforward as normal methods.

That is,

X.h

template<typename T>

struct X{

T a;

void yo();

};

X.cpp

#include ”X.h”

void X::yo() {} //error, which template’s yo does this definition instantiate ?

template<typename T>

void X<T>::yo(){} //works, but useless

main.cpp

#include “X.h”

int main() {

X<int> x{2};

x.yo(); //error

}

void X<T>::yo(){…} is useless here and x.yo() is a linker error. Because C++ compiles files individually, so when it complies main.cpp, it will get X.h’s contents and wait for definition of yo(), however when X.cpp is compiled, it doesn’t instantiate any instance of X so it doesn’t link void X<T>::yo() {} with anything, i.e., X.cpp doesn’t already have an instance and X<T>::yo() doesn’t know if it has to be linked to any instance.

There are 3 solutions to this:

* Forgo forward declaration and define the method in the struct/class or in the .h file itself.
* Use an .inl file, this file is called an inline file, and it basically extracts all its contents in place of declaration. So we move X.cpp’s definitions into X.inl file and #include “X.inl” in the X.h file (at the end of #include list) and it will work. The .inl file can only contain inline definitions, i.e., only variables and methods that are forward declared otherwise it may void ODR and cause compilation error.
* 3-file approach, we can still use the X.cpp, if we #include “X.h” and then also include “X.cpp” in main.cpp then the definition in X.cpp is applied to all instances used in the main.cpp file. This works but is non-standard, a slightly better alternative to #include “X.h” and #include “X.cpp” in a “templates.cpp”, that is

templates.cpp

#include “X.h”

#include “X.cpp”

template class X<int>; //or it is template struct X<int>; if X is a struct

template class X<double>; //same as above

And now use X.h anywhere, this is the same as using X’s instances in main.cpp and it will link the method definition in X.cpp with all instances but is better for compilation and linker speed.

template class X<int>; is called explicit template instantiation, this syntax causes the compiler to create an instance of a given template right then.

* + - 1. Class Template with Non-type parameter

Just like function templates, class templates can also use non-type params, these params are constexpr and are hence are required to be defined at compile time itself.

Class templates can use these as non-type params,

* An integral type
* An enumeration type
* A pointer or reference to a class object
* A pointer or reference to a function
* A pointer or reference to a class member function
* std::nullptr\_t
* A floating point type (since C++20)

For ex.:

#include<iostream>

template<int sz>

struct X{

int x[sz]{};

};

int main() {

X<5> x{}; //ok

int a;

std::cin>>a;

X<a> y{}; //error, sz must be constexpr

}

* + - 1. Functions inside structs

Structs can define functions within them too. These functions are called member functions or methods.

For ex.:

#include<iostream>

struct X{

int d{};

void yoyo() {

std::cout<<"yoyo"<<'\n';

}

void yo() {

std::cout<<d<<'\n'; //can access other members of X

yoyo();

}

};

int main() {

X k{2};

k.yo(); //Use the member selection operator to call member functions

}

One key difference between functions inside structs or classes and outside structs or classes is that they don’t need to be defined before to be used. This is because the struct or class object is built with ‘knowledge’ of all its members and then after initialization when a member function accesses other members it is referring to the object itself, i.e., d and yoyo() inside yo() up there are actually calling k.d and k.yoyo() and the object of X d is implicitly passed to yo(), and here the object d is called the implicit object.

The rest is continued in the class section as class and structs are identical (with a few differences).

* + - 1. Explicit Member Function Template Specialization

EFTS is also applicable to classes/structs when they use template or their methods use templates.

For ex.:

struct X{

template<typename T>

void yo(T a) {}

};

template<>

void X::yo(int a) {}

int main() {

X x{};

x.yo(3); //calls our specialized yo(int) instead of creating a new definition.

}

When there’s a template inside a class/struct, the class/struct itself is defined only once (unless the class/struct itself is also templatized) but the method is defined for each instance used, after compilation.

When the class/struct itself is templatized,

#include <iostream>

template <typename T>

struct X {

T a;

X(T v1) : a{v1} {}

~X() = default;

};

template <>

X<int>::X(int v1) : a{v1} {}

template <>

X<int>::~X() {}

int main() { X x{2}; }

This is the only case where redefinition of an already defined class/struct method definition is allowed because the class/struct isn’t actually defined when under a template, it is waiting to be defined by the compiler when types are passed to the template at compile time.

Class Template Specialization

We can go a step further and explicitly specialize entire class/struct as well.

For ex.:

template <typename T>

struct X {

T a;

X(T v1) : a{v1} {}

~X() = default;

};

template <>

struct X<int> {

int a;

X(int v1): a{v1} {}

~X() = default;

};

int main() { X x{2}; }

Access specifiers can be changed too for specialized classes/structs.

Partial Template Specialization

Works as described earlier, however it doesn’t work for methods, only for whole class/structs.

That is,

template<typename T, int sz>

struct X {

T a;

void pp() {}

};

template<>

void X<int,3>::pp() {} //ok

template<int sz>

void X<int,sz>::pp(){} //error

int main() {

X<int, 3> x{2};

}

We can do Partial template specialization for the whole class/struct here instead, for ex.:

template<typename T, int sz>

struct X {

T a;

void pp() {}

};

template<int sz>

struct X<int,sz> {

int a;

void pp(){}

};

int main() {

X<int, 3> x{2};

}

It also works similarly for Function Templates.

This could lead to a lot of repeated code so it is recommended to declare most of the stuff in a common base class instead.

* + - 1. Inheritance with Class Template and Partial Template Specialization

Inheritance is defined in the class section in this doc, but on the topic of partial template specialization for classes/structs I define inheritance in the mix here

We can inherit a templatized class like so

template<typename T, int sz>

struct X {

T a;

void pp() {}

};

template<typename T,int sz>

struct Y: public X<T,sz> {};

template<int sz>

struct Z: public X<int, sz> {}; //ok, Partial Class Template Specialization

template<>

struct K: public X<int, 3> {}; //error, Explicit Class Template Specialization doesn't work

int main() {

Y<int, 3> y{2}; //ok

y.pp(); //ok, pp is public

}

* + - 1. Class Template Specialization for Pointers and References

We can also explicitly specialize class templates for pointers and refs,

to do so

template<typename T, int sz>

struct X {

T a;

void pp() {}

};

template<typename T, int sz>

struct X<T\*,sz> {

T a;

void pp() {}

};

template<typename T, int sz>

struct X<T&,sz> {

T a;

void pp() {}

};

This also works with Partial Template Specialization,

template<typename T, int sz>

struct X {

T a;

void pp() {}

};

template<typename T>

struct X<T\*,3> {

T a;

void pp() {}

};

works.

* + 1. Arrays

These are also aggregate data types that can hold multiple values, but of same type.

For ex.:

int a[10] {}; //is an array

std::cout<< a[0]; //prints the first element

This is a fixed-size array as the size is defined at compile-time.

The initializer list here defines the default value for the 10 values of the array, 0 by default.

We access each member, called element, of the array with the subscript operator [ ] and this process is called indexing or subscripting. The range of index starts from 0 to n-1, where n is the size defined for the array. The subscript value must be integral, it is numerically converted to int if it is integral but not an int. Expressions and bools work too

a[2+3]; //works as a[5] is accessed

a[false]; //converted to a[0]

Any type can be made into an array.

The size must be a CTC, so either macro or constexpr or raw literal value. RTCs and runtime values result in error.

To initialize the values,

int a[10] { 1,2,4,5 }; //ok, the rest values are initialized to 0

int b[3] {1,2,4,5}; //error, more values in the initializer list than size of the array

int c[5] {}; //ok, all values are initialized to 0, called zero initialization

int d[5]; //uninitialized values

int e[] {1,2,4,5}; //ok, the compiler deduces the size automatically as length of initializer list,4

We can assign values to indices outside the range of the array and that is not an error, but leads to undefined behavior.

* + - 1. Passing to Functions and Returning from functions

C++ doesn’t pass arrays by copying each value, it implicitly passes the array by reference, so

#include<iostream>

void yo (int arr[3]) //syntax to define array in function param

{

arr[0]=5;

}

int main() {

int a[3] {1,2,3};

std::cout<<a[0]; //prints 1

yo(a);

std::cout<<a[0]; //prints 5

}

We can define the array param in the function as const int arr[3] to disallow modification.

Function param arrays can also let compiler deduce size, so int arr[ ] works too.

* + - 1. Size

aka length of an array. There are 3 ways,

#include<iostream>

#include<iterator> //from c++17

int main() {

int a[ ]{1,2,3};

std::cout<<sizeof(a)/sizeof(a[0]); //prints 3

std::cout<<sizeof(a)/sizeof(int); //prints 3

//c++17

std::cout<<std::size(a); //prints 3, unsigned int 3

//c++20

std::cout<<std::ssize(a); //prints 3, signed int 3

}

sizeof(a) returns the size of the whole object and each value has size given by either the type or any value of the object so we get no. of values.

size and ssize are more modern methods.

Neither of these methods work as expected inside a function on a passed array due to how arrays are passed.

* + - 1. Sort

Sorting means arranging all elements either in ascending order based on their value or descending order. Aggregate Data types like arrays can be sorted using custom sorting alrgorithms or pre-defined ones.

std::swap in #include<utility> swaps 2 variables, even array values. For ex.:

#include<utility>

int main(){

int x{2};

int y{3};

int a[3]{3,4,5};

std::swap(x,y); //x is now 3, y is now 2.

std::swap(a[2],a[0]); //a[0] is now 5, a[2] is now 3

}

std::sort in #include<algorithm> is a quick way to sort an ADT like an array.

#include<algorithm>

int main() {

int a[5] {2,45,12,1,3};

std::sort(std::begin(a), std::end(a));

}

sorts array a.

* + - 1. Multidimensional Array

An array consisting of another array is a multidimensional array.

For ex.:

int a[2][3] {

{1,2,3}, {4,5,6}

};

a[0][1]; //to access 2

This is a 2D array, it says array of size 1, and the element is an array of size 2.

Can be zero-initialized,

int a[2][5] { }; //ok

Unlike normal 1D arrays, 2D arrays can at most only omit the left length specification, the others need to be defined.

int a[][5] { {…}…}; //ok

int a[][] { {…}… } //error

* + - 1. Pointers of arrays

When a fixed-size array is used in an expression, it decays, i.e. implicitly converts, into a pointer. The pointer is the first element’s memory address.

For ex.:

#include<iostream>

int main(){

int a[3]{};

std::cout<< &a[0] <<” ”<< a;

}

Prints the same thing twice. So in this case both &a[0] and a return int\*.

This means if we do \*a then it prints the value of the first element.

And that we can assign a, which returns a pointer, to another pointer, so int\* p {a}; works.

There’s still difference between an int\* and an array’s int\*. This is visible using sizeof()

For ex.:

#include<iostream>

int main(){

int a[3]{};

int\* p {a};

std::cout<< sizeof(a)<<’\n’;

std::cout<< sizeof(p)<<’\n’;

}

prints 12 then prints 4.

sizeof() multiplies the size of int\* with no. of elements in a, so 4\*3 = 12.

Secondly, the address-of operator behaves differently for an array vs. an int\*.

For ex.:

#include<iostream>

int main(){

int a[3]{};

int\* p {a};

std::cout<< a; //prints something like 0xabcd

std::cout<< &a; //prints the same

std::cout<< p; //prints the same

//but

std::cout<< &p; //prints 0xnotabcd

}

This is because the type is different for &a vs. &p, &a’s type is int (\*)[5], &p’s type is int\*\*.

Or in other words, the address-of operator when used to get address of an array pointer, gets the same pointer, but a normal int\* gets the address where the int\* is stored.

This is why, the array param in functions is actually a pointer.

For ex.:

#include<iostream>

void pp(int a[]) {

std::cout<<sizeof(a); //prints 4

}

void pp2(int\* a) {

std::cout<<sizeof(a); //prints 4

}

int main() {

int a[3] {};

std::cout<<sizeof(a); //prints 12

pp(a);

pp2(a);

}

Arrays inside structs or classes don’t decay.

Arrays are laid out sequentially in the memory, so &a[0] returns the first address, then &a[1] will return the next object’s memory address and so on.

For ex.:

#include<iostream>

int main() {

int a[3] {};

int\* p{a};

using namespace std;

cout<< &a[0] << “ ”<< p; //prints 0x1234 and same

cout<< &a[1] << “ “ <<p + 1; //prints 0x1238 and same

}

This shows how arrays are treated internally in C++, and using just the first object’s pointer and size of the array we can iterate through it all.

* + - 1. std::begin() and std::end()

Both of these take an array as a parameter and return an iterator, which is a kind of pointer. begin() returns the iterator/pointer at the first object of the array and end returns the pointer/iterator exactly 1 after the array’s last element in memory. Defined in #include<iterator>

For ex.:

#include<iostream>

#include <iterator>

int main() {

int a[3]{};

auto beg{std::begin(a)};

auto end{std::end(a)};

while(beg!=end) {

std::cout<< \*beg;

beg++;

}

}

Prints all elements of a.

Because beg is a pointer, pointer arithmetic works on it and it points to the next object on increment, which as we know in array is the next element.

Whilst beg and end are iterators, the behavior is the same with normal pointers, so we can also do

auto beg{a};

auto end{a+3};

* + - 1. std::count\_if()

Syntax:

std::count\_if(<begin pointer of type T>, <end pointer of T>, <function\_name>);

The function should accept a param of type T and return a bool.

This method takes the start and end pointer of an array, and a function that can accept each element of the array, then it passes the elements one by one to the function and the function either returns true or false for each element. This method counts the number of true’s and returns that integer.

* + - 1. C-Style String

aka null-terminated String. Is an array of chars. Even std::string is implemented on top of C-Style strings. By default, strings in C++ are C-Style strings unless denoted as std::string with the literal s.

For ex.:

char s[ ] { “yo”}; //is a c-style string

The length of s is 3 and not 2, because C-Style strings are automatically null-terminated, that is a character, ‘\0’ is appended, this character is special as it indicates the line ending. Here string within “…” is actually a const char \* being copied into s.

Since this is an array,

s= “yo”; //doesn’t work

s[0]= ‘y’

s[1]= ‘o’; //works

Care must be taken to not overwrite the last character, which is the null character and if overwritten will lead to undefined behavior when this string is read, such as when used in cout, cout will keep printing the characters from the buffer until it comes across any null character from adjacent memory.

The list initializer for a char array/C-Style string with const char \* automatically inserts a null character at the end, but it is possible to initialize each character much like a normal array, in which case it doesn’t null-terminate the string automatically.

For ex.:

char s[] {‘h’,’e’,’l’, ‘l’,’o’}; //size 5

char s2[] {“hello”}; //size 6

s is non-null terminated char array/C-Style string whilst s2 is null terminated.

Input

Taking input using cin is disregarded as no. of characters in input isn’t fixed.

Getline is recommended,

For ex.:

#include<iostream>

#include<iterator>

int main() {

char s[5]{};

std::cin.getline(s, std::size(s));

}

getline only takes given no. of characters -1 and last character is null.

Other useful C-Style String Manipulator Methods

Most of them are defined in #include<cstring>

Copy

std::strcpy(<dest char array>, <source char array>);

//will error if dest. not big enough.

std::strlen(<char arr>); //returns length

strlen is different from size, as it returns no. of chars before null char. whereas size returns total no. of chars the object can hold.

strcat: concatenate

strcmp: compare 2 strings

* + - * 1. C-Style String Symbolic Constants

Basically the raw representation of what string literals really are.

That is,

int main() {

const char\* s {“yooo”};

}

Aka pointer to string literal, this string literal is exactly what we have in other places too, any string between “…” is a CS3C, or simply, a string literal.

The compiler automatically applies null character at the end here.

But this is a bit different from a normal C-Style String that is stored as a char array. This is a symbolic constant, and we can’t modify it, C++ treats CS3Cs as static duration objects and not as local duration objects.

Furthermore, C++ may also deduplicate multiple similar CS3Cs.

For ex.:

const char\* s1 {“yo”};

const char\* s2 {“yo”};

at runtime may point to the same memory address.

These are static duration, so this is the only case where we can return raw literal from functions,

const char\* yo() {

return “ooga booga”;

}

works and we don’t have to worry about the CS3C being destroyed at the end of the scope

* + - * 1. cout and C-Style Strings

cout automatically changes its behavior when char\* is passed to it.

For ex.:

#include<iostream>

int main() {

const char\* s{“yo”};

char s1[]{“yo”};

int a[]{1,2};

std::cout<< a; //prints 0x1234

std::cout<< s1; //decays to char\*, then prints “yo”

std::cout<< s; //already a char\*, prints “yo”;

std::cout<< &a; //prints 0x1234

std::cout<< &s1[0]; //prints garbage

}

cout doesn’t print char\*, it instead accesses it and prints that. This is why when it comes across &char\*, it tries to print the address as string but it finds some other value stored there.

* + - 1. std::array

It is defined in #include<array>, it aims to simplify fixed-size arrays.

std::array is recommended over raw fixed-size arrays.

For ex.:

#include<array>

int main() {

std::array<int, 5> arr{1,2,3,4,5};

std::array arr2{1,2,3}; //also ok since C++17

std::array<int> arr3{…}; //error

auto arr4 {std::to\_array<int,4> ({1,2,3,4})}; //ok since C++20

auto arr5 {std::to\_array<int>({1,2,3})}; //ok since C++20

auto arr6{std::to\_array({1,2,3,4})}; //ok since C++20

}

That is, basically there are 2 ways to create std::array objects and we can let compiler deduce type and size for normal std::array or to\_array function, or let compiler deduce size for to\_array. And we can use list initializer for all of them. If the provided no. of elements in the list initializer are less than the size of the std::array, they are initialized to 0.

Declaration and assignment also work, but type and size must be given at the time of declaration,

For ex.:

std::array<int,5> arr;

arr={1,2}; // rest elements are zero-initialized

Access works like normal arrays.

The at() function

std::array supports bounds checking so accessing invalid indices doesn’t lead to undefined behavior, to use it we use the at() instead of subscript []operator to access an element.

For ex.:

std::array arr{1,2,3};

arr[4]=2; //undefined behavior

arr.at(4)=2; //throws std::out\_of\_range at runtime

at() is slower than subscript operator but provides bound checking.

<object>.size() to get the size of the std::array. The returned size is of type size\_type(a nested type within std::array) which is an alias for size\_t, both are unsigned.

For ex.:

std::array arr{1,2,3};

for(std::array<int,3>::size\_type index{0}; index< arr.size(); ++index) {…}

//as size\_type is an alias for size\_t, we can use that instead

for(std::size\_t index{0}; index< arr.size(); ++index) {…}

RBFL works on std::array.

std::array doesn’t decay, so we can treat it as a normal type to pass around functions.

For ex.:

#include<array>

#include<iostream>

std::array<int,2> yo(const std::array<int,5>& arr) {

std::array<int,2> arr2{};

arr2[0]=arr[0];

arr2[1]= arr[1];

return arr2;

}

int main() {

std::array arr{1,2,3,4,5};

std::array arrX{yo(arr)};

std::cout<<arrX[0];

}

works as expected. std::array requires size to be defined at compile time so even the function return has to define it, here arrX doesn’t need to define size is because it is known at compile time that yo() will return an array of size 2.

We can also use templates to let the compiler generate the function definitions and allow the same function to be called with any sized array.

For ex.:

#include<array>

template<typename t, std::size\_t s>

void yo(const std::array<T,s>& arr) {

…

}

int main() {

std::array arr{1,2,3,4,5};

yo(arr);

std::array arr2{2.0,3.0};

yo(arr);

std::array arr3{1,2}; //generates a new definition

yo(arr);

}

With size as non-type template parameter and T as template parameter we let the compiler generate the definition. The size is also used in overload resolution so whilst arr and arr3 have the same type, the sizes are different so different definitions are created for them.

Array of structs

std::array can be over any type, but with struct initializers it needs to be initialized a slight bit differently. This is because std::array is defined as a struct itself with a normal C-Style array inside it.

For ex.:

#include<array>

struct S {

int x;

int y;

int z;

};

int main() {

std::array<S,3> arr {

{1,2,3},

{4,5,6},

{7,8,9},

}; //is an error

std::array<S,3> arr2;

arr2[0]= {1,2,3};

arr2[1]= {…};

arr2[2]= {…};

//ok

std::array<S,3> arr3 {

{

{1,2,3},

{4,5,6},

{7,8,9},

}

}; //ok

}

The reason arr doesn’t work is because std::array is itself a struct, the first element to arr {1,2,3} is taken as the value for the first element inside the std::array, the first element is indeed the C-Style array so it is ok, but the rest elements have no members to attach to. Anywho, the behavior we want is to attach all the 3 list initializers to the C-Style array, so we use another pair of braces above them to map them to the first element of the std::array struct.

* + - 1. Iterator

An iterator is an object meant to traverse through all members of a container, such as all values of an array.

The logic behind iterators is simple, to provide a uniform interface to traverse through any container’s members.

The most basic iterator is a pointer,

For ex.:

#include<array>

#include<iterator> //for std::begin(), not arr.begin() and std::end()

int main() {

std::array arr{1,2,3,4};

auto beg{std::begin(arr)}; //or auto beg{&arr[0]}; //or arr.begin()

auto end{std::end(arr)}; //or auto end{beg+ std::size(arr)}; //or arr.end()

while(beg!=end) {

… //use \*beg

++beg;

}

}

C++ provides the iterator library that generates iterators for arrays, these iterators are different for each type, for std::array, the iterator type is std::array<T,size>::iterator.

Iterator Invalidation

Iterators behave much like pointers, so if we call begin() or end() then the iterator might be pointing to a correct address and this is copied. But now if the value resizes, this copy can’t be affected and hence the iterator may be pointing to an invalid address, this is called iterator invalidation.

* + 1. Dynamically Allocated Memory

Normal variables are allocated memory on stack by the compiler, this is fixed region in the memory. But some variables can dynamically allocate memory at runtime, this memory is reserved on the heap.

To allocate a single object’s worth of memory on the heap, we use the new keyword.

For ex.:

new int; //allocates 1 int worth of memory on the heap and returns the address.

int\* p {new int};

and now we can use p like a normal int\*.

The reason this type of memory is special is because the program itself is responsible for (de)allocating memory instead of the compiler doing it for the program. This memory is reserved to be shared on the OS so it is passed around when cleared.

Initialization

We can initialize the dynamic variable with a value,

int\* p {new int(5)}; //direct init.

int\* p1{new int{5}}; //uniform init.

The latter is recommended.

It is recommended to use this pattern for dynamic memory allocation,

int\*p {nullptr};

if(!p)

p= new int;

If p is nullptr only then shall it be allocated.

This is because when a memory is dynamically allocated, but the address is simply ‘forgotten’ by the program, it still remains allocated and cannot be freed until the program ends and the OS clears all memory associated with the program. This is called a memory leak.

To deallocate the memory used by a pointer we use the delete keyword,

int\* p{new int};

delete p; //now the memory address of p is deallocated. p is still pointing to the address.

p=nullptr; //ok

When we deallocate, the OS can choose to give it to some other program or not, nevertheless the value there may or may not remain the same. Accessing a deallocated pointer is an undefined behavior, and the pointer pointing to a deallocated memory address is a dangling pointer. Deallocating an already deallocated memory address is also undefined. However, when the pointer is nullptr, calling delete on it does nothing.

Deallocation is necessary for all dynamically allocated memory, otherwise the memory may remain allocated but never used up by any other program or this program itself, causing a memory leak.

new can fail

It is possible the OS doesn’t have enough memory on the heap to give, in this case dynamic allocation can fail and a bad\_alloc exception is thrown. We can either handle this exception or use std::nothrow constant.

For ex.:

int\* p {new (std::nothrow) int}; //initializes to nullptr if bad\_alloc

if(!p) {

std::cerr<<”Failed allocation”;

}

It is advised to use fixed-size arrays or compile-time variables instead of dynamic ones as they are less error-prone, faster to access and easier to manage. They are faster because of various optimizations the OS and CPU makes at runtime, such as caching which is more effective for variables on the stack.

* + - 1. Dynamically Allocated Array

Arrays can be dynamically allocated as well. To do so we use new T[] and delete[].

For ex.:

int length{5};

int\* arr {new int[length]{}};

arr[1]=2; //etc. use like normal array.

delete[] arr; //to deallocate

arr=nullptr;

allocates dynamic sized array on the heap, the memory addresses are adjacent just like normal arrays, delete[] then deletes all the memory addresses of the array. The reason delete doesn’t require size is because new[] internally keeps track of the allocated size, this size is inaccessible to the programmer.

Calling simple delete on dynamically allocated array only deallocates the first address and the rest are left as is, which is an undefined behavior and causes memory leak.

A dynamic array is much like a normal array that decays into a pointer.

To initialize, from C++11 we can use initializer list. It works like initializer list for normal arrays.

int\* arr{new int[5]{1,2,3}};

* + - 1. Dynamically Allocated Multidimensional Array

We make use of pointer to pointers here.

In 2D arrays, if the 2nd size is constexpr, i.e., a CTC, then we can use a normal Dynamically Allocated Array.

For ex.:

constexpr int cols{2};

int rows{3};

int (\*arr)[cols] {new int[rows][cols]}; //works

auto arr2 {new int[rows][cols]}; //can also use auto

If the rightmost size isn’t a CTC, then we use pointer to pointers.

For ex.:

For 2D

int rows{10};

int cols{5};

int\*\* arr {new int\*[rows]};

//Then set each pointer to a dynamic 1D array,

for(int i{0}; i<rows; ++i) {

arr[i] = new int[cols];

}

//now

arr[0][2]; //works

//And to deallocated

for(int i{0}; i<rows; ++i) {

delete[] arr[i];

}

delete[] arr;

Here we basically create a dynamic array where each element has the type T\*. Then each element gets a pointer to another dynamically allocated array. And we have our 2D Dynamically Allocated array.

Alternatively we can just use a 1D array that is flattened, i.e., if the size of 2D array is 2x4 so 8 values, then we can simply have a 1D array with size 8 and translate 2D indices into 1D to access each element, in this specific case we know every 4 elements a row is traversed, so the element at 5 is actually [1][0] in 2D and so on.

For ex.:

int get1DIndexFrom2D(int row, int col, int totalCols) {

return (row\* totalCols )+col;

}

is a simple function that translates 2D indices into 1D.

* + - 1. std::vector

Much like std::array is a replacement for fixed-size arrays, std::vector is a replacement for dynamic sized arrays.

Defined in the #include<vector> header.

std::vector is recommended over raw dynamically allocated arrays.

std::vector doesn’t need size declaration at all, it automatically manages the size.

For ex.:

std::vector<int> arr; //ok

std::vector<int> arr2= {1,2,3}; //ok since C++11, initializer list copy initialization

std::vector arr3 {1,2,3}; //ok since C++17, uniform initialization from C++03 and CTAD from C++17

std::vector is a class and not a struct, so it has many constructors. Direct initialization (automatically invokes the right ctor) or manually invoking a ctor also works.

For ex.:

std::vector<int> arr(5); //direct initialization, invoking std::vector<int>(size\_t size) ctor, allocates 5 elements and inits them to default value of 0 for int

std::vector<int> arr2(3,4); //invokes std::vector<int>(size\_t size, int defaultValue), so allocates 3 elements and inits them to 4

std::vector<int> arr3(3, {1,2,3}); //invokes std::vector<int>(size\_t size, <initializer list>), allocates 3 elements and assigns them these values.

//Now their manual ctor counterparts

std::vector<int> arr4{ std::vector<int>(5) };

std::vector<int> arr5 { std::vector<int>(3,4 )};

std::vector<int> arr5 { std::vector<int>(3,{1,2,3})};

vector is automatically resized. So, for ex.:

std::vector arr{1,2,3};

arr = {1,2,3,4,5}; //replaces the internal array with this array and resizes to size 5

arr = {1,2,3}; //same

Access elements with [ ]operator or at(). But if the element is out of bounds, vector doesn’t resize or return default value, it behaves like normal arrays and gives undefined behavior with [ ] or throws with at().

vector has self-cleanup. So it automatically deallocates when the object goes out of scope.

size(), works exactly like it did for std::array. However, the size() returns the no. of elements allocated not the actual dynamic allocation size.

That is,

#include<vector>

#include<iostream>

void printSize(std::vector<int>& arr) {

std::cout<<arr.size();

}

int main() {

std::vector arr{1,2,3};

printSize(arr); //prints 3

arr= {};

printSize(arr); //prints 0

}

std::vector<bool>

This has a special implementation defined for it so it is optimal when dynamically allocated bitsets are required.

For ex.:

std::vector<bool> arr{true,false, 0, 1, 1};

Iterator Invalidation in practice

Here we can see an invalidated iterator,

For ex.:

#include<vector>

#include<iterator>

int main() {

std::vector v{1,2,3};

auto it{v.begin()};

v.erase(it); //removes the element at iterator from the vector

++it; //undefined behavior

\*it; //also undefined

}

In the case of erase(), it actually returns a new iterator pointing to the next element or end().

it= v.erase(it); //works

Resizing, Reallocation and Capacity

Vector can be resized, if the new size is > old size then old values remain as-is but values at new size are initialized to default value of the type. If the new size < old size then the excess values are trimmed. Resize actually modifies just the size but if the capacity is not as big then that is incremented to match the size, this is known as reallocation.

For ex.:

#include<vector>

#include<iostream>

void printSize(std::vector<int>& arr) {

std::cout<<arr.size();

}

int main() {

std::vector arr{1,2,3};

printSize(arr); //prints 3

arr.resize(4);

printSize(arr); //prints 4, and 1,2,3,0 are the elements.

}

The capacity is the no. of addresses allocated by the vector, the size is the no. of addresses that have been used. The capacity is guaranteed to be atleast equal to size no. of addresses and is generally greater, it is automatically managed so the programmer doesn’t need to (de)allocate capacity for the vector.

For ex.:

#include<iostream>

#include<vector>

int main(){

std::vector v{1,2,3};

std::cout<<v.capacity()<<’\n’; //returns an int denoting the capacity, would be atleast 3 here

std::cout<<v.size()<<’\n’; //returns an int denoting the size, prints 3

v.resize(5);

std::cout<<v.capacity()<<’\n’; //prints 5

std::cout<<v.size()<<’\n’; //prints 5

}

resize modifies the size to be 5, if the capacity was already 5 then reallocation doesn’t happen and only the size changes to 5 with the new values being default initialized, if the capacity was <5 then reallocation occurs and the new size is 5 whilst the new capacity may be 5 or greater.

It is recommended to not manually resize as it is a computationally expensive task, reallocation is an even more expensive task, and the capacity may be large enough to hold all the values already.

For ex.:

std::vector v {1,2,3}; //capacity could be already 5

v= {4,5,6,7}; //no change to capacity, size changes though

v = {1}; //same, and we don’t need to worry about left-over elements, i.e., v[1] would be an error here.

The size/length is what the [ ]operator and at() use, not the capacity. Accessing vector through [ ] or at() doesn’t resize it.

Reserve

Using .reserve(<int>); we can set the capacity to a given int to avoid reallocations, this is especially useful when our vector is resizing/reallocating a lot in the program. By reserving a given capacity upfront, the performance penalty incurred with resizes can be reduced.

Vector as stack

Vectors have 3 methods that make them behave like stacks, the .push\_back() method to push elements at the back, .pop\_back() method to remove the last element and .back() to retrieve the last element.

These methods can cause resize.

Reallocations grow exponentially, the capacity just needs to be as large as the size but Visual Studio reallocates by increasing the capacity by 1.5x so if the cap is already 5 then the next reallocation may increase it to 7, GCC may increase it to 10 as it grows by 2x there.

* 1. Type Deduction

aka Type inference. In C++ we can use the auto keyword (much like let in javascript) to let the compiler automatically deduce the type of a variable based on a value.

For ex.:

auto x{2}; //x is int

However the type must be inferred at initialization and it must inferable at compile time itself.

So

auto x; //error

auto also removes top-level constness. So

const int x{2};

auto y{x}; //y is of type int

For string literals the inferred type is char\* and not string, we have to use the string literal to denote the type it should take, literals can also be used to denote types of other values.

* 1. Literal

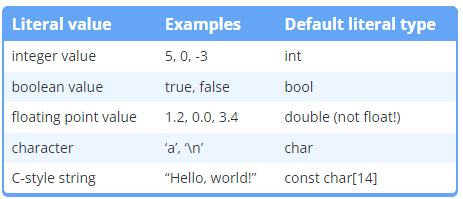
aka literal constants. These are raw values and are fixed so they never change throughout the program’s lifecycle.

For ex.:

int x{5};

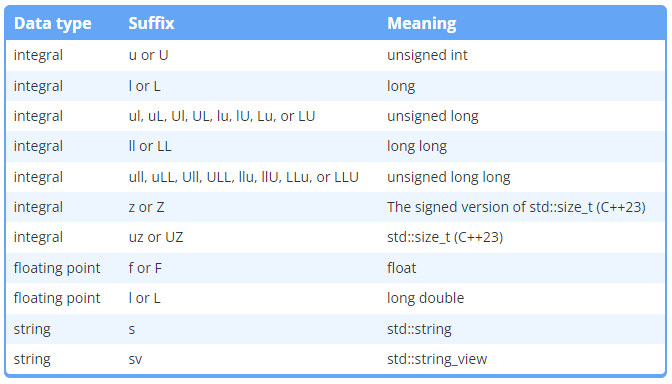
here 5 is a ‘literal’ and is copied into x. But just like variables, literals have a type and a value. The only difference is that they can’t be directly modified at runtime, the variable that copies a literal can however change its own value.

For ex.:



* + 1. Types

The type of a literal can be defined explicitly as well, to do so we add a suffix character indicating its type. These are the suffixes we can use in C++



They aren’t case sensitive.

* + 1. Digit Seperators

We can break long digits into smaller pieces using ‘.

For ex.

int x{1234’56’78};

stores 12345678 in int but it is more readable now.

1. Functions

It’s a reusable sequence of statements.

The most important function in any C++ code is the main() function, this function is the entry point of execution for the program that is executed and the program terminates at the end of main or if prematurely exited by either the user or a statement.

A function calling a function is called a caller.

Syntax:

<return type> <identifier>(<args>) //this line is called the function header

{

… //function body

}

The identifier is a unique name. Every function needs a return value type in C++.

if function’s name is abc then its simply called with

abc();

* 1. Nested Functions are not supported in C++.

That is

int abc() {

void bb() {

}

}

is an error.

* 1. Return values

int abc() {

return 0;

}

Here 0 is the being returned from the function. return is a special keyword that immediately halts the execution of a function and sends a value or nothing back,

return;

is valid and returns nothing. That is also to say, functions can only return a single value.

Every function except void returning functions need to return a value of return type in c++, if there is no return from non-void returning functions then undefined behavior may occur.

However, every function can return, even void functions, void can only use return;

Yes, main() needs to return too, the integer returned by main signals the OS the status of the program. 0 means success, and all other values indicate failure.

C++ defines 2 macros to make it standardized, the EXIT\_SUCCESS which is equal to 0 and EXIT\_FAILURE.

#include <cstdlib> // for EXIT\_SUCCESS and EXIT\_FAILURE

int main()

{

return EXIT\_SUCCESS;

}

However main() is an exception to the ‘every function must return’ as if it doesn’t return despite having return type of int then it implicitly returns 0.

* 1. Void functions

Can use return; but it’s not required unless we want to halt execution somewhere in the function.

Can’t be used where a value is expected.

For ex.:

void abc(){ }

int x {abc()};

is an error.

* 1. Parameters and Arguments

Functions accept values being passed to them at call time, for this to be possible they need to define their ‘parameters’.

For ex.:

void abc( int x, int y) {

…

}

and we provide values for these parameters at call site like so

abc(2,3);

Here 2 and 3 are called the ‘arguments’ of function abc. 2 is passed to x of abc and 3 is passed to y of abc.

Parameter is definition, and argument is the actual value.

* 1. Forward Declaration and definition

In C++ this is an error

#include <iostream>

int main()

{

std::cout << "The sum of 3 and 4 is: " << add(3, 4) << '\n';

return 0;

}

int add(int x, int y)

{

return x + y;

}

because unlike other languages, functions need to be declared before they are used from top of the file to the bottom of the file. There are exceptions to this, such as when functions are a part of a class or struct, then they can be called before or after in the code.

C++ expects functions outside objects to be declared before being used. So in the above example, add function being defined before main() would solve the error.

Alternatively, C++ also allows for ‘forward declaration’, that is

int add(int x, int y); // forward declaration of add() (using a function declaration)

int main() {

…

}

int add(int x, int y) {

…

}

tells the compiler add is declared here (above) and defined later.

Forward declaration doesn’t require variable names, just variable types,

int add(int, int); //is valid

Not defining a forward declared function is a linker error (but only when the function is called somewhere).

* 1. Inline Functions

Normally, functions in HSL are stored in a given address and the executor has to jump to that address after storing the current state in registers to execute that function and then return back to the stored state and continue. This has a small performance overhead and can be avoided if we use inline functions. Inline functions are those whose bodies can replace the call site invocation, called inline expansion. The downside is that the size of the compiled code is larger and a single function can lead to multiple expansions further worsening the size and compile time.

For ex.:

int min(int a,int b)

{

return (a<b ? a : b);

}

int main()

{

std::cout<<min(2,3);

}

Here min() call would get replaced by the body, so only (…) would be left in main().

The inline keyword could be used to denote if a function should be expanded but modern compilers ignore it and decide by theirselves.

* 1. Constexpr function

A function that can be solved at compile time can use this keyword to be optimized away.

For ex.:

constexpr int add(int a, int b) {

return a+b;

}

int main() {

constexpr int c {add(1,2)};

std::cout<<c;

}

works and c is replaced with 3.

These functions are implicitly inline.

However, even non-constexpr values can use this function and there it is not inlined.

The compiler determines if a constexpr function must be evaluated and inlined at compile time if the caller is constexpr too.

A function can determine if it was called at compile time using is\_constant\_evaluated() brought by C++20.

#include<type\_traits>

constexpr int yo() {

if(std::is\_constant\_evaluated())

{ … }

else { … }

}

* 1. Consteval functions

These functions only execute at compile time and are hence always inlines and always evaluated at compile time.

For ex.:

consteval int add(int a,int b){

return a+b;

}

int main() {

int x{add(2,3)};

int y{};

cin>>y;

add(y,y); //error

add(x,x); //ok

}

Constexpr funcs can be forced to be CTC too using a neat trick with consteval func.

consteval auto doStuff(auto f)

{

return f;

}

constexpr int add(int a,int b)

{

return a+b;

}

int main() {

int x{doStuff(add(2,3))};

}

works and is always ran at compile time (given add function is compile time evaluatable), whilst we can use add even at runtime unlike a consteval which by itself can only be used at compile time.

This works because doStuff needs CTC value which add can do so compiler forces it to do.

* 1. Static function

Just like variables can have static duration, so can functions, in which case they are file scoped.

For ex.:

static void pp(){

...

}

will only be accessible throughout the file (Internal Linkage only) and also have a static duration.

* 1. Function Overloading

This is a feature in C++ that allows multiple functions with same name to exist as long as their parameters are different, either in types or in number of params or both. Type aliases and constness (const, references and volatile) doesn’t differentiate parameters.

However for member functions (inside classes or structs), the constness of params does matter.

For ex.:

#include <iostream>

int yo(int x) {

return 2;

}

int yo(long x) {

return 3;

}

int pp(int x) {

return 4;

}

int main()

{

std::cout << yo(3L) << '\n';

std::cout << pp(3L) << '\n';

return 0;

}

prints 3 then 4.

The process by which compiler determines which overloaded function to call is called overload resolution.

Functions with ellipses are also considered different.

void yo(int x);

void yo(int x,…);

are considered different functions.

Machine Language doesn’t understand functions, nor function overloading as we see in C++, so the linker generates appropriate instructions to use different functions and the way C++ compiler allows linker to see different functions with same name (but overloaded) is through name mangling, which gives these functions different names. NM isn’t the same as minification in javascript which shortens all identifiers, it is primarily only used to differentiate overloaded functions. [ChatGPT]

* + 1. Overload Resolution

Overload resolution finds the intended function, the result could be any of the 3.

1. Found a matching argument function.

When overload exists and for given args, there exists a function with exactly the same args then that is used.

For ex.:

void yo(int x) {…}

void yo(char c) {…}

yo(2); //will call int func

Since there is a func with exactly the same type params, it is used and overload has been resolved.

These are called exact matches and they also work for trivial conversions, i.e., when value remains the same but type changes.

For ex.:

void yo(const int x){ … }

void yo(double x) {…}

int x{2};

yo(x); //cals the const int func

Since only the constness is changed, it is considered a trivial conversion.

1. No matching argument function

When no func signature matches the call and trivial conversions won’t find the right func, overload resolution tries numeric promotion first and then it tries numeric conversion.

For ex.:

void pp(int x) {…}

void pp(double x) {…}

pp(‘a’); //calls int func

pp(2.5); //calls double func

pp(2); //calls int func

Promotion is based acc. to the priority list of promotion. So if arg has type T(char here) and it can be promoted to first T1 (int here) and then T2(double here) where T2 has the higher priority than T1 then T1 gets the call. Recall that promotion only goes up so no data loss should occur.

If promotions won’t resolve a call then conversion is done, for ex.:

void pp(double x){…}

void pp(std::string x){…}

pp(‘a’);

Here since char doesn’t exist it is promoted into double so double takes over, promotion has higher precedence than conversion.

If the type T has user-defined conversion to another type then they are attempted to match a call. This has even lower precedence than numeric conversion.

Finally if a function still hasn’t been found, a function with ellipses is looked for since ellipses accept variable args. These have the lowest precedence.

1. Ambiguous argument functions

None of that shit works = Ambiguous call error at compile time.

It is recommended to use casting or generics to define the correct function at call-site to be more clear.

* 1. Default arguments

C++ supports these so functions can have default args.

For ex.:

void pp(int x, int y=2) {…}

pp(2); //works.

Can be multiple but default args have to be defined at the right end of param list for function signature.

And C++ doesn’t support named args nor skipped args. For ex.:

void pp(int x, int y=2, int z) {…}

pp(2,,3)

wouldn’t work, firstly because non-default args have to be at the left so they need a vaue and secondly because we have to specify argument by position in C++ and empty ,, don’t work.

* + 1. For forward declaration

We can’t have default args in both forward declaration and definition. Only 1 of them must provide the default value.

For ex.:

void pp(int x, int y=4);

void pp(int x, int y) {…}

//and

void yo(int x, int y);

void yo(int x, int y=4) {…}

works. In the former case all definitions would get the same default value if arg is not provided and in the latter, all definitions could choose to allow default args and also define what that value would be.

* + 1. For function overloading

Overload resolution works like normal but things like these are a no-go.

void pp(int x);

void pp(int x, double y=2.5);

void pp(int x, int y=3);

pp(2);//error, ambiguous call.

pp(2,5); //calls int func

pp(2,5.2); //calls double func

* 1. Pass arguments by value and by reference

Functions params can be passed values by value, meaning the value itself is copied by the params or be passed by reference, meaning the param acts as an alias of the value being passed.

That is

void pp(std::string x, std::string& y, const std::string& z) {

…

}

int main() {

pp(“yo”,”x”,”y”); //error, non-const LVR needs L-Value, “x” is an R-Value

std::string s1{“yo”};

const std::string s2{“yo”};

pp(“yo”, s1, s2); //ok

pp(“yo”, s2, s1); //error, non-const LVR can’t refer to const L-Value

pp(“yo”, s1, “yo”); //ok, const LVR can refer to R-Value

}

pp’s x param accepts value so any argument to it is copied to x. y and z are LVRs so they only act as aliases and they don’t take their much memory, nor do they copy the passed value so the operation is cheaper compared to x.

Since y is a non-const LVR, it can modify the value of referent so s1 in main() will be affected if we change the value in pp().

The reason we don’t pass all values by reference is because unlike normal LVRs, there is a small cost (memory space) associated with references when used in function params and unlike normal LVRs these aren’t optimized away. These LVRs take up a small space (~2 words of memory, word = size of memory address) and then use that space to store the address of the referent.

So if the referent takes <=2 words of memory and has no setup cost (runtime) then it is more optimal to pass by value instead.

Setup cost is no. of the startup methods that a type runs after being initialized, like if we copy a class then the class would run its constructor and the methods defined in that ctor, so it does have some startup cost.

To check the word size of any type we can use this #define function

#define isSmall(T) (sizeof(T) <= 2 \* sizeof(void\*))

For setup cost, we have to manually measure the performance but a good rule of thumb is that all user-defined types have or will have setup cost.

* 1. Pass Arguments by Address

A 3rd way to pass function params, here we pass the memory address of objects instead of the value or the objects theirselves. This is enabled by using pointers.

For ex.:

void pp(const std::string\* s) {…} //use the string raw pointer s

int main() {

std::string s{“yooo”};

pp(&s);

}

It is recommended to check if the pointer param is null when a function accepts pointer types.

Just like the references, pointers change the value of the object. And RPCO doesn’t just like const reference doesn’t.

Pointers take up memory as well, and when passing by reference the memory address is copied from the caller to the param which then takes either 4 bytes or 8 bytes.

Default args can be used with pointer params,

void pp(const std::string\* s= nullptr) {…}

null is the only default value they accept though.

We can also pass by reference of pointer, int\*&.

In reality, there’s only pass by value. Pass by reference is implicitly pass by address and actually the address is passed and handled automatically, and memory address is an actual value so essentially there’s always some value being passed to functions.

* 1. Return by reference and Return by address

Functions in C++ can return by value, but also by reference and address (as we know, there is only value no address or reference, this is more of a wrapper over value).

For ex.:

int pp (…) {} //normal value return type

int& pp1(…) {

static int someVal{2};

return someVal;

} //reference return type

const int& pp2(…) {

static const int someVal{3};

return someVal;

} //const reference return type

int\* pp3(…) {

static int\* p {nullptr};

return p;

} //return by address

Return by reference or address must make sure the object whose reference/address is being returned must outlive the function’s lifetime and match the caller’s lifetime otherwise the object will be destroyed with the function end and the returned ref/address will lead to undefined behavior.

Return by reference can lead to odd situations like these

int& x(){

static int someVal{0};

++someVal;

return someVal;

}

int main() {

int& x1{x() }; //gets reference to x, with value 1

int& x2{x() }; //gets 2 and also sets x1 to 2

std::cout << x1 << x2 <<'\n'; //prints 22

x1 = 5;

std::cout << x1 << x2 <<'\n'; //prints 55

int x3{x()}; //copies the reference

int x4 {x()};

std::cout << x3 << x4 <<'\n'; //prints 67

}

Since it is a reference, all instances having that reference get modified if the function modifies the returned reference. This is why it is preferred to use const references and/or make value copies.

We can simply return the reference parameters as reference, this works as the duration of these params matched the caller anyway.

Return by address works similar to return by address but the caller must check if the returned int\* is not nullptr.

* 1. Type Deduction for functions

We can also use auto for function return types (C++14) and also for parameters (C++20), although for function parameters, auto uses a feature called function templates from C++20 which is like normal templates but automatic for this case.

For ex.:

#include <iostream>

#include<typeinfo>

auto yo(auto x) {

std::cout << "yo says "<<typeid(x).name() << '\n';

return x;

}

int main()

{

std::cout << "main says "<< yo<double>('a')<< '\n';

return 0;

}

prints “main says yo says double” then “97”. Here <> explicitly defines the type of the parameter as double and ‘a’ is promoted to double. If we simply call yo(‘a’) it will remain a char.

auto drops the top-level constness of return type.

For ex.:

const int yo() {

return 2;

}

int main() {

auto x{yo()}; //x is of type int not const int as constness is dropped

const auto y{yo()}; //ok and type is const int

}

Similarly auto drops the reference.

For ex.:

const int& yo() {…}

int main() {

auto x{yo()}; //x is of type int

auto& y{yo()}; //y is of type const int&

const auto& z{yo()}; //applies const and reference to auto’s deduced type, z is const int&

}

In the case of y, auto deduces type to be int and applies reference, and for reference const int& is low level const so it is kept and hence we get const int&. That is, auto only gets int and const int has top-level const, so const is dropped and int remains, auto& gets int& and const int& is low-level const so const is kept.

auto only removes the Top-Level Constants.

auto doesn’t drop pointers

int\* y2(){…}

int main() {

auto x{y2()}; //x is of type int\*

auto\* y{y2()}; //same

}

auto and auto\* have a bit of difference.

auto when deducing pointer type, deduces the type to be T\*, which is expected, however auto\* deduces the type to be T and then applies pointer to that type, which is the same as auto if the value is indeed T\* but an error if it is not.

For ex.:

int\* y2(){…}

int main() {

auto x{\*y2()}; //x is int

auto\* y{\*y2()}; //error as int is not assignable to int\*

}

const auto and auto const are the same thing for auto

int\* yo(){…}

int main() {

const auto x{yo()}; //x is of type int\* const

auto const y{yo()}; //same

const auto const z{yo()}; //error, can’t reapply const

}

const here says applies const to whatever type is deduced, so T\* gets applied const and becomes T\* const.

pointer to const and const pointer with auto

For ex.:

const int\* const yo(){…}

int main() {

auto x{yo()}; //ok, x is of type const int\*

auto\* y{yo()}; //same

const auto\* z{yo()}; //same

}

In all of these cases, const int\* means const is low-level and hence it is kept.

To get const int\* const type, we can use auto\* to make

const auto\* const p{…};

Similarly, auto\* const would make T\* const type.

* 1. Function Pointer

Function definitions are actually an L-value type, where the type is defined by the params and return type of the function. Since they are an L-value, they are stored in memory. The () operator simply indicates the executor to go to the memory address and start executing instructions there.

To get the address of a function, we simply avoid () operator.

For ex.:

#include<iostream>

void foo () {

}

int main() {

std::cout<<foo<<’\n’; //if the cout treats the address as an int, explicitly cast the address to a void\*

std::cout<< reinterpret\_cast<void\*>(foo)<<’\n’;

//reinterpret cast forces the type to change but let the value remain the same if the type can hold it.

}

Pointer

To create a function pointer,

<return type> (\*<someName>)(<params>);

For ex.:

int (\*p)(); //is a function pointer p for a function that returns int and has no params.

The brackets are necessary, otherwise if we do int \*p(); it will be taken as a forward declaration for a function.

To initialize we use the same list initializer as for normal variables,

For ex.:

int yo(int a, char c) {

return 2;

}

int main() {

int (\*p) (int. char) {yo};

int (\*p2) (int. char) {&yo}; //same thing, we just explicitly denote that we are taking yo’d address

p=&yo; //assignment works like normal

int (\*p3)() {nullptr}; //function pointers can be assigned nullptr as well.

}

Here yo’s type is int (\*)(int,char). This is the function type.

To actually call the function,

(\*p)(5, ‘a’); //explicit dereference and call the function

p(5,’a’); //implicit dereference and call the function

Both work and we can use either.

One interesting thing to note is that all functions are passed around as pointers, so

void yo(){…}

int main() {

(\*yo)(); //works, calls yo()

}

is valid.

Default args don’t work with function pointers, this is because the way C++ handles default args and params is that it adds the default arg at call-site at compile-time but function pointers are a runtime concept and hence can’t have the args defined at call-site at compile time (because the pointer can change which function it points to) so default args don’t work with function pointers and we need to provide them explicitly with function pointers.

For ex.:

void yo(int c=4) {

}

int main() {

void (\*p)(int){yo};

(\*p)(4); //ok

(\*p)(); //error

void (\*p2)(){yo}; //error, void (\*)(int) in unassignable to void(\*)()

}

These are just like normal pointers, so sending and returning function pointers from functions work.

void yo(int (\*p)(int,double)) {…} //ok

//and called with

int aye(int a, double c) {

return 2;

}

int main() {

yo(aye); //ok

}

Since functions are a type, they can be passed as a value too, but in this case they are implicitly passed as pointers.

That is,

void yo(int (\*p)(int,double)) {…}

//and

void yo(int p(int,double)) {…}

//are the same thing

and both can be called using either p(…) or (\*p)(…). The latter definition of p is only valid for function parameters, everywhere else it is seen as forward declaration.

This also shows why a function yo(…) can be called with (\*yo)(…). It’s because functions are always seen as pointers.

* + 1. Default parameter

Just like normal values, function pointers can also take default values in parameters.

For ex.:

void yo(int (\*p)(int,int) = someotherfunction) {…} //ok

* + 1. Type Aliases for Function Pointers

Type alias can be used here as functions are a type.

For ex.:

using MyYo= void(\*)(int);

void yo(void (\*p)(int)) {…} //can be now written as

void yo2(MyYo p) {…} //same thing but with alias

* + 1. auto

auto can be used to simplify the function pointer declaration.

For ex.:

void (\*p)(int) {…}; //is equivalent to

auto p2 {&somefunction};

* + 1. const

Function pointers can be made const too, this is a high level const so the pointer can’t point to another function.

For ex.:

void (\*const p)(…) {…}; //ok

const int (\*p2)(…) {…}; //also ok but it means the return type is const int

* + 1. std::function

Defined in #include<functional>, this is a type to simplify function pointers, works exactly the same way though.

For ex.:

#include<functional>

void yo(int a,char c){…}

int main() {

void (\*p)(int,char){&yo}; //is equivalent to

std::function<void(int,char)> p2 {&yo};

}

p2 can only call its function with p2(), i.e., implicit dereference.

CTAD can infer the type for std::function from C++17.

* 1. Recursion

When a function calls itself, its called a recursion.

For ex.:

int yo(int a){

if a<=0:

return 0;

return yo(a-1);

}

int main() {

yo(1);

}

calls yo with 1, which calls yo with 0 and now this yo returns 0, and the yo with 1 returns 0 to main(). The if condition in a recursive function is called a recursive termination condition, without it the recursion goes for infinite times, crashing the program when the call stack reaches its size limit.

Iteration vs. Recursion

Almost always a recursive function algorithm can be laid out in using a loop, loops are better than recursion because the call stack doesn’t have to add a function (and its parameters) in the stack and hence increase the size as much which leads to better performance and more efficient use of the call stack.

The downside to iterative algs is the fact that recursion is generally easier to implement and is good enough for most cases.

* 1. Main()’s Parameters

main() is the entrypoint to the program, and it accepts parameters too, these parameters are passed arguments from the command line itself, called command line arguemtns.

Main() has 2 definitions for accepting parameters:

<void/int> main(int, char\*[]);

and

<void/int> main(int, char\*\*);

both are same.

The first param is the no. of args supplied and the latter is each arg.

For ex.:

Say our file is called main.cpp and compiles to main.exe, then if we call

./main.exe abc 3

in the command line, then

#include<iostream>

int main(int argc, char\* argv[]) {

std::cout<<argc<<’\n’; //prints 3

std::cout<<argv[0]<<’\n’; //prints “C:/somewhere/main.exe”

std::cout<<argv[1]<<’\n’; //prints “abc”

std::cout<<argv[2]<<’\n’; //prints “3”

}

argv[0] is the file’s path.

Each arg is space separated, but if we enclose a string with “” then all values in it make just 1 arg.

* 1. Ellipsis

This operator allows a function to accept any number of arguments.

Defined in #include<cstdarg>.

For ex.:

#include<iostream>

#include<cstdarg>

double summar(int count, …) //Atleast 1 parameter needs to be normal when ellipsis is used, ellipsis needs to be the last param.

{

double sumVal{0};

std::va\_list list; //This type is required to access the args passed to …

va\_start(list, count); //initializes va\_list list with the args passed, the 2nd arg to va\_start is the last non-ellipsis param of the function to denote where the ellipsis args start.

for(int arg{0}; arg<count;++arg) {

sumVal += va\_arg(list, int); //va\_arg gets the arg from the va\_list as the given type

}

va\_end(list); //cleanup the va\_list

return sumVal;

}

int main() {

std::cout<<summar(4, 1,2,3,4)<<’\n’; //works, prints 10

std::cout<<summar(2, 3,5)<<’\n’; //prints 8

}

As we can see ellipsis requires a lot of boilerplate.

va\_\* are function macros defined in cstdarg. The way this works is basically like pointers and array, the args are passed like an array, va\_list works like a pointer and tracks the elements in this array, va\_start sets the pointer va\_list to the first ellipsis element in the arglist of the function. Then va\_arg simply gets the value at this pointer and moves this pointer forward. va\_end then clears this pointer up. Calling va\_start again after va\_end would reset the pointer and we can go at it again.

Type checking is suspended with ellipsis

That is, the compiler doesn’t know the type of any arg in the ellipsis. va\_arg simply assumes the given arg is of given type, if it isn’t then undefined behavior may occur. And the program doesn’t even know the no. of args. It is all upto the programmer. This is why ellipsis are to be avoided.

From C++11 we have variadic templates, then from C++17 we can use fold expressions. Both of these are better alternatives to raw ellipsis.

* 1. Fold Expression

These make use of parameter packs and reduce parameter packs into singular values by applying binary or unary operations on them. Works from C++17.

For ex.:

template<typename... Args>

bool all(Args... args) { return (... && args); }

bool b = all(true, true, true, false);

Takes each arg, &&s it with the next one and repeats. b will be false here. Similarly we have || operator.

template<typename... Args>

int sum(Args&&... args)

{

return (args + ... + (1 \* 2)); // OK

}

works too.

The basic syntax is

(pack OP …) //apply OP operator to each arg of the pack, start from left, like (1 + 2 + 3), start from right so first do 3+2 then 5+1

(… OP pack) //same but start from right, so for (1+2+3), first do 1+2 then 3+3

(pack OP … OP init) //apply OP operator and then at the end apply this normal value, like (1 + 2 + a) where 1+2 came from pack, goes LTR

(init OP … OP pack) //same but first solve normal value then apply operator, also LTR

OP can be any of the 32 operators: + - \* / % ^ & | = < > << >> += -= \*= /= %= ^= &= |= <<= >>= == != <= >= && || , .\* ->\*

* 1. Anonymous Function

These functions don’t have a static duration like normal functions so they are part of the call-stack and take up memory there. They are supposed to be used for defining a quick function to process values without proper definition. Aka Lambda Expressions or closures.

The syntax is like so

[<capture list>] (<param list>) -> <return type> {<body>};

The capture list allows us to define values from the parent scope and allows the lambda to use these values, param list is like normal functions and return type indicates the return type. All of these are optional, so the most basic lambda is

int main() {

[ ] { }; //a lambda with no capture list, param, return type or body.

}

The return type can be deduced automatically for lambdas so we can omit them anyway.

We can also store a lambda but we need to use auto or std::function,

For ex.:

int main() {

auto myLambda {[ ](int x) {

return 2+x;

}

};

//and called like a normal function

myLambda(3); //returns 5

}

We need to use auto because the type of a lambda is only generated at compile-time, they are like functions but aren’t exactly functions, they are an object called functor with the () operator overloaded.

std::function also accepts a lambda but applies a small overhead so it is a bit slower than using lambdas, however std::function is required if we need to pass our lambda to functions.

For ex.:

#include<functional>

void yo(int x, const std::function<int(int)>& f) {

//call f(<int>) whenever

}

int main() {

auto myLambda {[ ](int x) {

return 2+x;

}

};

yo(2, myLambda); //works

std::function {[ ] (int x) {…}}; //also works, CTAD works here

}

From C++20, even functions can use auto f thanks to abbreviated function templates.

Return type can be deduced automatically, but it requires all returns to be of same type.

Lambdas are great for trivial tasks because they are easier to implement but it is recommended to use normal functions when a lot of reuse is required.

* + 1. Generic Lambda

Lambdas can accept generic params too.

For ex.:

int main() {

auto myLambda {[ ](const auto& a, auto b) {

return a+b;

}

};

myLambda(3, 5); //returns 8

myLambda(3.0, 5); //returns 8.0

}

This lambda is called a generic lambda.

Just like function templates, each generic lambda resolves to a specialized lambda for a given type at compile time. That is,

#include<iostream>

int main() {

auto myLambda {[ ](const auto& a, auto b) {

static int yo{0};

std::cout<<yo++;

return a+b;

}

};

myLambda(3, 5); //returns 8, prints 0

myLambda(3.0, 5); //returns 8.0, prints 0

myLambda(3.0, 5); //returns 8.0, prints 1

myLambda(3, 5); //returns 8.0, prints 1

}

* + 1. Capture list

By default the lambda only has access to global identifiers and entities that are known at compile time, and entities with static duration. The capture list allows us to define values it should capture from its parent scope to use in itself.

For ex.:

int main() {

int x{2};

auto myLambda {[x]() {

return x+2;

}

};

myLambda(); //returns 4

x=3;

myLambda(); //returns 4 even so

}

The capture list is resolved at definition time of lambda, in runtime. That is, unlike parameters, the captures are only initialized once. The lambda makes a copy of the values in the capture list with the same name. So when the executor reaches the lambda definition it puts whatever value of x is into the lambda in a variable x and calls it done, this x isn’t affected by the outside x.

By default, the captured value is cloned into a const variable, meaning we can’t modify it. So x++ in the above example wouldn’t work.

To remove this constness, we use the mutable keyword like so

int main() {

int x{2};

auto myLambda {[x]()mutable -> int {

x++; //ok since x is mutable

return x+2;

}

};

myLambda(); //returns 5

myLambda(); //returns 6, x is still 2 everywhere since it is not affected

}

The return type is still optional, but I placed it there to show where it is with the mutable keyword. One thing to note is that the captured variables are cloned and te clone belongs to the lambda, like a static object, so whenever lambda modifies the cloned variables it affects it for itself permanently just like a static object.

Captured values can be passed as references or pointers, they are non-const by default and affect the main variable too. They are still being cloned like normal variables but as we know references and pointers are an alias to the memory address so that is copied here instead.

For ex.:

int main() {

int x{2};

auto myLambda {[&x]() {

return x+2;

}

};

myLambda(); //returns 4

x=3;

myLambda(); //returns 5

}

We can manually make x const with const &x so it disallows changes on x inside the lambda.

We can mix const, non-const, reference and values in the capture list.

Care must be taken with reference and pointer types that they are valid in the lambda, since the capture copies the value, in case of pointer and references, the memory address is copied, if this memory address is dangling and the lambda is called it will lead to undefined behavior.

Default capture

Captures all variables in the context. We use ‘=’ to capture all by value, and ‘&’ to capture all by reference. We can even mix in normal capture variables too, in which case those variables are captured as defined but every other variable is captured using default capture.

For ex.:

int main() {

int x{2};

int y{3};

int z{4};

auto myLambda {[=,&x]() { //capture x as reference, and all others as values

return x+z+y;

}

};

myLambda(); //returns 9

}

Default capture has to appear first in the capture list.

Declaring new variables

We can declare new variables in the capture list too.

For ex.:

int main() {

int x{3};

auto myLambda {[y{2\*x}]() {

return y+2;

}

};

myLambda(); //returns 8

}

The type is auto deduced.

* + 1. Copy

Lambdas can be copied.

int main() {

int x{3};

auto myLambda {[x]()mutable {

return x++;

}

};

myLambda(); //returns 3

auto lambda2{myLambda};

lambda2(); //returns 4

myLambda(); //returns 4

}

The captured and cloned values are also copied, so both lambda2() and myLambda() have an x in them with value 4 when the copy is made, and this copy doesn’t affect anything else.

This could be problematic if the lambda is being passed to a function and its internal state is not being modified even if it is passed as a reference.

For ex.:

#include<functional>

#include<iostream>

void yo(const std::function<int()>& fn) {

std::cout<<fn()<<’\n’;

}

int main() {

int x{3};

auto myLambda {[x]()mutable {

return x++;

}

};

yo(myLambda); //prints 3

  yo(myLambda); //still prints 3

}

This is because std::function makes a copy of our lambda internally, and uses that in reference instead so fn has a new copy and not the actual lambda.

To solve this we can force pass our lambda as a reference using std::ref which wraps the type in std::reference\_wrapper, both defined in #include<functional>. std::ref passes T as &T so any changes made to T are applied here as well.

For ex.:

#include<functional>

#include<iostream>

void yo(const std::function<int()>& fn) {

std::cout<<fn()<<’\n’;

}

int main() {

int x{3};

auto myLambda {[x]()mutable {

return x++;

}

};

yo(std::ref(myLambda)); //prints 3

  yo(std::ref(myLambda)); //prints 4 now

}

std::ref only works if the parameter also accepts reference, it only enforces references from the call-site. So

#include<iostream>

#include<functional>

void yo(int x) {

std::cout<<x++<<'\n';

}

int main() {

int x{3};

yo(std::ref(x)); //prints 3

yo(std::ref(x)); //still prints 3

}

That’s why this doesn’t work as expected.

* 1. Trailing Return Type

As we saw with lambdas, we can use a different syntax for defining functions.

This is the trailing return type syntax

for ex.:

auto yo(int) -> int {return 0;}

is a valid function syntax in C++. Here the return type of yo() is int, but this syntax can only be used when the return type is auto on the left, the return type on the right can be omitted as well. It’s generally used for brevity but the normal syntax should be preferred anyway.

1. Standard Library Algorithms

C++ is famous for its algorithms and data structures operating on these algs.

Generally used with arrays these provide a lot of predefined optimal functionality,

such as

Inspection: View data in a container, such as searching or counting.

Mutation: Modify. Sort or shuffle.

Facilitation: Generate a result based on values of the members, such as reduction.

There’s

std::sort to sort an array

For ex.:

#include<algorithm>

#include<array>

bool greater(int a, int b) {

return a>b;

}

int main() {

std::array arr{1,2,3};

std::sort(arr.begin(), arr.end(), greater); //sorts arr in descending order

std::sort(arr.begin(), arr.end(), std::greater{}); //from C++17

std::sort(arr.begin(), arr.end(), std::greater<int>{}); //from C++11

}

std::greater is a type, not a function so it needs {}. It also does what our greater function did but is predefined.

std::find to find an element in an array

std::find\_if to find the first element that satisfies a condition

std::count counts the no. of occurences of an element

std::count\_if

std::adjacent\_find to find if 2 adjacent elements return true for a condition

etc.

C++20 introduces Ranges, this helps in performance even more than iterators.

1. OOP

Object-Oriented Programming. Unlike normal programming where we simply define functions and statements, called procedural programming, this paradigm focuses on “objects”, i.e. entities in the programming with properties and behaviors defined for them.

To represent the ‘blueprint’ or design of an object we use Program-Defined Types, such as classes or structs.

Classes and structs are identical in c++.

* 1. Class

Defined just like a struct.

class ABC {

public: //required for here

int a{};

int b{};

int e{};

void yo() {…}

}; //semicolon needed for PDTs

int main() {

ABC yo{1,3,4}; //initialized just like structs

yo.yo(); //ok

}

Just like a struct, the members are called with member selection operator on the object and the object is implicitly passed (in \*this).

By convention, Class names should be PascalCase and member variables should begin with m\_ prefix.

* + 1. Both Classes and Structs can have member types/nested types

These are the types that are declared inside the class/struct. Type Alias is also allowed.

For ex.:

class A{

public:

using IT= int;

IT a{};

};

int main() {

A a {1}; //ok

A::IT x{2};

A b {x}; //ok

}

Just like nested structs, we can access alias’ and type’s as the class acts as a kind of namespace.

Classes can also nest classes/structs. So can structs.

* + 1. Access Specifiers

These determine who will have access to members of a class/struct.

For ex.:

class A {

private:

int x{};

};

int main () {

A a {1}; //error

}

This is because of the access specifier ‘private’. These are declared like that and can be multiple per class/struct.

An access specifier applies access permissions to all the members below it, and if another access specifier comes below, then that one applies its access permissions to the members below it.

There are 3 of these,

private: The members are only accessible by the members inside class/struct or by friend functions.

public: Anyone can access the members.

protected: Members are accessible to members inside class/struct, friend functions and by members inside inheriting class/struct and their friends.

classes are ‘private’ by default, meaning their members are private access specified by default. structs are ‘public’ by default.

It is a good practice to have public methods but private data members.

For ex.:

#include<iostream>

class A{

private: //not required, since it is private by default

int x{};

public:

void printX() { //method is public

std::cout<<x<<’\n’; //can access x here

}

};

int main() {

A a{12}; //error

A b{}; //ok

b.printX(); //ok, prints 0

}

Access Specifiers work on a per-class basis and not per-object basis.

That is, no matter which object, the access specifier limits access as defined on the class.

For ex.:

class A {

private:

int x{};

public:

int getX(const A& a) {

return a.x; //ok

}

}

int main() {

A a{};

A b{};

b.getX(a); //works

}

As we can see, even if the object is different, the access specifier is only limiting which class can access it, not which object can.

* + 1. Constructors

As we have seen, aggregate initialization can initialize data members but with access specifiers they can’t do so. We use special methods called constructors to initialize an object.

Ctors can take any number of params but their only rules are that they can’t return anything (not even void) and must have same name as the class.

For ex.:

class X{

private:

int x{};

public:

X() { //default ctor

x= 2;

}

};

int main() {

X x{}; //implicitly calls the default ctor. Called value initialization.

X y; //same, called direct initialization

}

Both invoke the default ctor if it exists, but both have downsides, if the ctor doesn’t exist then value init. would zero-initialize all public data members, and direct initialization wouldn’t even do that so if the data members don’t have default initialization and there is no ctor, direct initialization of the class object will leave them undefined.

A default ctor is the class’ primary ctor.

Ctor must be public to be accessible.

Ctors can be overloaded

class X{

private:

int x{};

public:

X() { //default ctor

x= 2;

}

X(int value) {

x= value;

}

};

int main() {

X x; //calls default ctor

X y{3}; //calls ctor with param, called list initialization

X z(3); //same, still called direct initialization

X c = 3; //copy initialization, finds the ctor that can take this value and passes it.

}

Copy init. should be avoided as it makes a temporary object and then copies it into our object.

List initialization is the recommended way to initialize objects in C++. As we see, it automatically calls the appropriate ctor with just the arguments given, this applies everywhere, even on function returns.

For ex.:

class X{

private:

int x;

public:

X(int value) {

x=value;

}

};

X yo() {

return 3+5; //or return {} for the parameterless ctor. or {3,4} for 2 param ctor and so on.

}

int main() {

X a{yo()}; //works

}

yo() returns X type and passes it the value 8. We can omit the uniform initialization brackets if a single param ctor exists for T, otherwise we need to use the uniform initialization brackets.

Implicit constructor

If there is no ctor in a class, a public ctor is automatically generated, called implicit ctor. This ctor has no param and empty body.

If there is a ctor with params and no ctor without params, then the ctor with params becomes the default ctor, and the implicit ctor isn’t generated either.

For ex.:

class X{

private:

int x{};

public:

X(int value) { //default ctor

x= value;

}

};

int main() {

X x{}; //error, value is required

X y; //same

X z{2}; //ok

}

We can use default params in the ctor as well, in which case it works like normal functions, i.e., if we don’t provide the value, it uses the default value.

We can manually make C++ generate a default parameterless empty body ctor,

For ex.:

class X{

private:

int x{};

public:

X() = default;

X(int value) {

x= 2;

}

};

int main() {

X x{}; //ok

}

The default keyword is shorter than writing a default ctor and better indicates the intentions.

If a class has a data member that is a class, then it calls its default ctor right before its own ctor. This is different from how the fundamental data type members are treated as these are class-type members.

For ex.:

#include<iostream>

class Y{

public:

int z;

Y() {

z=2;

std::cout<<”Y()”<<’\n’;

}

};

class X{

public:

int x;

Y y;

X() {//notice x is not getting initialized

std::cout<<X()<<’\n’;

}

};

int main() {

X x;

x.x; //undefined behavior

x.y.z; //ok returns 2

}

prints Y() then prints X(). And class-type members are auto-initialized but fundamental data type members are not.

Ctors don’t create new objects, the object is created before the ctor is even invoked. Then the ctor simply performs whatever statements are defined in it. The params of a ctor define who can create an object. We can call ctor again but it is unrecommended as ctor might lead to another object eing created which will get automatically discarded.

* + - 1. Member Initialization List

We use this special syntax to initialize const data members and other members that require value at initialization (like references).

For ex.:

class X{

private:

const int x; //const int x{} inits x to 0 and prevents assignment

const int& y; //const int& y {} is an error

const int a[5]; //arrays can be initialized too

SomeClass B;

public:

X(int& value): x{3}, y{value}, a { 1,2,3,4 }, B{1,2} {

//called a member initialization list

//if we had done x=3 here then that would have been assignment not initialization, hence it would have been an error

}

};

int main() {

int a{3};

X x{a}; //ok

}

[This is kind of like in c#]

They all use uniform initialization, hence a is initialized as if normally and the 5th value is initialized to 0 and B’s ctor with 2 params is called.

This is recommended over assignment inside the ctor, as this directly initializes the data member.

In a class/struct, data members are initialized in the order of declaration, not in the order of member list initializer.

To define method body for ctor with member list initialization,

class X {

public:

int a;

X(int v1);

~X() = default;

};

X::X(int v1): a{v1} {} //works

int main() { X x{2}; }

We saw something similar in Explicit Member Function Template Specialization.

* + - 1. Non-static member initialization

Data members can be initialized with default values if the ctor doesn’t value for them.

For ex.:

class X{

private:

int x {2}; //non-static member int x is initialized to 2

int y {}; //This is just default zero-initialization

public:

X(int s){

y=s; //This overwrites default or NSM initialization

}

};

int main() {

X x{2};

}

Precedence list for member initialization (in other words, what overwrites what),

NSMI or Default Initialization

Member Initialization List

Ctor Assignment (assignment inside ctor body).

That is, if ctor assignment is given for a member then that overwrites all the other initializations.

* + - 1. Delegating Constructors

A ctor can invoke another ctor. This helps remove redundant behavior.

For ex.:

class X{

private:

int x;

public:

X(int value): x{value} {

}

X() : X{0} { //delegating ctor, calls the ctor which accepts 1 param

}

};

int main() {

X x{}; //ok, calls the default ctor which calls the 1 param ctor

X y{2}; //ok, calls the 1 param ctor

}

Delegating ctors call the other ctor first, then execute their own body.

* + - 1. Reset an object

Rather than defining a method that resets the data member values manually, we can do this

class X{

private:

int x;

public:

X() {

x=2;

}

void reset() {

\*this = X{}; //creates a new object of X and then does a memberwise copy into this object

}

};

int main() {

X x{};

x.reset();

}

* + - 1. new Object

Just like new allocates memory on the heap for a type and returns its memory address, also performing initialization if defined, new can do the same for classes as well.

For ex.:

class X {

…

};

int main() {

X\* x {new X{1}}; //allocate X object’s worth memory, initialize an X object in it which has a 1 param ctor, and return memory address

delete x; //to deallocate

x= nullptr; //good practise

}

* + - 1. Copy constructor

When a class object is being initialized with another object, then this requires a ctor that accepts the object of its own class. By default, if we don’t explicitly define a copy ctor, the compiler automatically adds a public copy ctor which does memberwise initialization, i.e., it copies each member from the other object into this object.

For ex.:

class X{

private:

int x;

public:

X(): x{0}

{

}

};

int main() {

X a{};

X b{a}; //works

}

Here a default copy ctor is automatically defined and it copies the x from object a into a new instance and passes this instance back to b.

We can explicitly create a copy ctor too,

for ex.:

class X{

private:

int x;

public:

X(): x{0}

{

}

X(const X& a): x{a.x} {

}

};

int main() {

X a{};

X b{a}; //ok, calls our copy ctor

}

The param must be a const reference of the object and can’t be directly the value type X. This is because if that’s allowed then the copy ctor would need to be called for copying from call-site to function param, then that would require copy ctor to be called again and the function param would require the copy ctor again and so on leading to an infinite recursion.

To disable copy ctor, we can simply move an explicit copy ctor into private access specifier.

Copy ctor Elision

When we have an anonymous class object being used to initialize a class object, then it requires 2 steps, initializing the anonymous class object and then copying it into the new class object but the compiler may use an optimization called copy ctor elision where it will directly initialize the new object with the args of the anonymous class lambda instead.

For ex.:

class X{…};

int main() {

X x{ X{0} };

}

Here X{0} is an anonymous class object, and through elision the compiler sees the entire statement as X x{ 0 }; by omitting the anonymous class object.

Up till C++17, elision is not mandatory and this is why the copy ctor is required to be public. But from C++17 most cases are covered and mandatory so the copy ctor isn’t always required.

Copy Initialization

This type of class object initialization calls the copy ctor.

For ex.:

class X{…};

int main() {

X x(2); //calls the copy ctor X(const X&)

}

This is why uniform list initialization is recommended as it avoids calling the copy ctor which produces a needless copy.

This type of initialization implicitly occurs when we pass our object by value to a function.

* + - 1. Converting Constructor

By default, C++ treats all constructors as implicit conversion operators.

For ex.:

class X{

private:

int x;

public:

X(int v1, int v2): x{v1+v2}

{

}

};

X yo() {

return {2,3};

}

int main() {

X a{yo()};

}

works, because when we return {2,3}, C++ looks for a ctor that accepts 2 int params and finds that there is one. In this case it uses that ctor to implicitly convert into to type X.

Ctors eligible to be used as implicit conversion operators are known as converting constructors.

To disallow a ctor to be used as implicit conversion operator, we can use the explicit keyword which requires explicit use of the ctor to use it.

For ex.:

class X{

private:

int x;

public:

explicit X(int v1, int v2): x{v1+v2} //make this ctor explicit

{

}

explicit X(int v1): x{v1} {

}

};

X yo() {

//return {2,3}; //error

//return X{2,3}; //ok

return static\_cast<X>(2); //explicit casting also ok

}

int main() {

X a{yo()};

}

Methods can be made explicit too.

Similarly we have the delete keyword to completely disallow a ctor/method from being used.

For ex.:

class X{

private:

int x;

public:

explicit X(int v1, int v2): x{v1+v2} //make this ctor explicit

{

}

X(int) = delete;

};

int main() {

X a{2}; //error

}

* + - 1. Move Constructor

Just like move assignment, the move ctor is responsible to move the content from the source object into this object.

For ex.:

#include <iostream>

#include<utility> //for std::move

class X {

private:

int\* x;

public:

X() : x{new int{0}} {}

X(const X& a) : x{new int{\*a.x}} { //Copy Ctor

std::cout << "Copy Ctor" << '\n';

}

X(X&& a) noexcept : x{a.x} { //Move Ctor

a.x=nullptr;

std::cout << "Move Ctor" << '\n';

}

X& operator=(X&& obj) { // Move assignment, notice the lack of const

if (&obj == this) // if they are the same address

return \*this;

delete x;

x = obj.x;

obj.x = nullptr;

std::cout << "Move completed" << '\n';

return \*this;

}

X& operator=(const X& obj) { // Copy assignment

if (&obj == this) return \*this;

\*x = \*obj.x;

std::cout << "Copy completed" << '\n';

return \*this;

}

~X() {

delete x; // even if nullptr delete x, works

}

};

X yo() { return X{}; }

int main() {

X x1{};

X x2{};

x2 = yo(); //prints Move Completed

x2 = x1; // prints Copy Completed

X x3{std::move(X{})}; //prints Move Ctor

X x4{x1}; //prints Copy Ctor

}

std::move() is necessary as otherwise X{} is elided into x3 in which case we can’t demonstrate move ctor.

We can disable copying by marking copy ctor and copy assignment with delete.

Move with std::swap

This leads to infinite recursion and must hence be avoided.

That is,

std::swap(\*this, obj);

whilst it seems simple, it invokes move on \*this, which invokes move on itself and so on.

Rule of 5

The copy ctor, copy assignment, move ctor, move assignment and destructor must all share the same status, i.e. either deleted or defaulted or explicitly defined.

std::move

It is a simple function defined in #include<utility> that static casts its object as an R-Value (returns R-value) meaning that the move operation is guaranteed on the object.

Care must be taken when std::move is invoked on an L-Value as C++ guarantees normal R-Values are temporary and are hence OK to be moved since their state won’t be seen anymore but L-Values may be used later, but C++ doesn’t give any guarantee for that case and the object may be in an unspecified state so we mustn’t use a std::moved L-Value.

std::move\_if\_noexcept

This function behaves similarly to std::move, but decides if move or copy should be performed. std::move is a bit dangerous, if the move ctor or assignment op. is non-noexcept and hence doesn’t have a strong exception guarantee then it is possible it may fail, and when it does both the objects will be in incorrect state. This is not a problem with copy because both objects exist anyway and only the values are copied but here this object is deallocated and the other object is modified.

move\_if\_noexcept simply returns an RVR when the move ctor for a type is noexcept or if there is only move ctor. Otherwise it returns a copyable LVR and hence invokes the copy operation for the type.

* + 1. Destructor

Dtor is just like a ctor, that it is automatically invoked. However a dtor is invoked when the object of the class goes out of scope or goes belly up for whatever reason, and its main purpose is to perform any cleanup such as if the class allocated some memory on the heap, this is where we deallocate it. Fundamental types and other classes/structs don’t need their dtor called in a dtor, as it would be called automatically.

Just like a ctor it has to follow some rules, same name as class but prefixed with tilde (~) operator, 1 dtor per class, no params and no return type.

For ex.:

#include<iostream>

class X {

private:

int\* p;

public:

X(int& f): p{&f} {

std::cout<<”Ctor”<<’\n’;

}

~X() { //destructor

delete p;

p=nullptr;

std::cout<<”Dtor”<<’\n’;

}

};

int main() {

{

int p{2};

X x(p); //prints “ctor”

} //prints ”dtor”

std::cout<<”main()”<<’\n’; //then prints “main”

int a{2};

X\* x{new X{a}};

delete x; //delete calls the dtor of the object

x = nullptr;

}

Resource Acquisition Is Initialization

RAII is a C++ compiler technique applied to automatic duration resources (like entities that are non-dynamically allocated), this ties the lifetime of a resource to its scope. That is, the resource can only be used as long as its scope is valid, after it it is automatically destroyed.

This concept should be called Scope-Bound Resource Management according to other programmers.

For classes/structs, it means that their dtors are automatically called once the scope of the class/struct object is up.

This is a bit different from how identifiers’ lifetime is handled, as we know it is dictated by their duration and not scope, but because of RAII or SBRM, it is dictated by their scope as well.

* + 1. \*this

As we know when a class’ object calls any method, and that method accesses any other class’ member, C++ knows which object to look into for the member. This is due to C++ implicitly passing the object into the method.

For ex.:

class X {

private:

int x;

public:

X(int value): x{value} {

}

void setX(int value) {

x= value; //works

}

};

int main() {

X x{2};

x.setX(3);

}

Here we can see setX knows which object’s data member x needs to be modified, but how exactly does it know ?

This is because during compilation

x.setX(3);

//becomes

setX(&x, 3);

and the definition of setX becomes

void setX(X\* const this, int value) {

this->x=value;

}

This is only done to non-static member function.

That is, the call-site passes the address of the object, the non-static member function gets a new parameter called this of type class pointer with high level const and all members accessed inside the body get converted to use the pointer this. All of this happens during compilation automatically.

This applies to the ctor and dtor as well.

We can explicitly use this inside the methods too, but it harms readability.

For ex.:

class X {

private:

int x;

public:

X(int value): x{value} {

}

void setX(int value) {

this->x= value; //works

}

};

int main() {

X x{2};

x.setX(3);

}

This does the same thing, the compiler just knows it will pass this later anyway.

Chaining function calls using \*this

To do so,

class X {

private:

int x;

public:

X(int value): x{value} {

}

X& add(int value) {

x+=value;

return \*this;

}

int getX() {return x;};

};

int main() {

X x{2};

x.add(2).add(3).add(4); //works, but returns X&

x.getX(); //returns 11, same if we just add .getX() above

}

As we know \*this represents the object’s memory address, so it’s duration is obviously longer than the function, so we can return it from a method. We also know X\* passed to the method is a reference, so using this knowledge we can return \*this which means X& from a method and now we can call another method on it.

* + 1. Class Methods can be forward declared.

We can forward declare a method inside a class body, and then define it outside it.

The syntax for defining a function outside is

<return type> <class name>::<method name>(<params>) {<body>}

If there’s no return type then it’s omitted, such as in ctor and dtor.

For ex.:

#include<iostream>

class X {

private:

int x;

public:

X(int value);

X& add(int value=2);

int print();

~X();

};

X::X(int value): x{value} {

//stuff

}

X& X::add(int value) {

x+=value;

return \*this;

}

int X::print() {

return x;

}

X::~X() {

//stuff

}

int main() {

X x{2};

std::cout<<(x.add(3).add(4).add().print())<<'\n'; //works

}

For default parameters, we don’t need to redefine them just like with normal forward declared functions. As for access specifiers, it doesn’t matter, no matter the access specifier on the function, if it is forward declared, it can be defined outside.

The convention is to define the class in a header guarded header .h file with forward declared methods and define the methods in a .cpp file. Then #include the header everywhere with the .cpp file being enabled for linking.

Member functions inside the class are considered implicitly inline whereas the methods outside are treated as normal functions so ODR applies to them fully.

Putting definition into .cpp file allows the same benefit as with forward declared function’s definitions, i.e., any change to the method definitions only require recompilation of the .cpp file and not the whole program which would be required if the class definition also included the method definitions.

This is also why external C++ libraries give a header file and a precompiled .cpp file, this makes the cpp file not require compilation and we can simply give it to the linker whilst using the .h file everywhere.

* + 1. Const class objects

A class object can be made const with the const keyword.

For ex.:

class X{

public:

int x;

public:

X(int value): x{value}

{

}

X(): X{0} {

}

void setX(int value){

x=value;

}

int getX() {

return x;

}

};

int main() {

const X x{};

const X y{2}; //ok

x.getX(); //compiler error as well

x.setX(2); //compiler error

x.x=2; //same

}

When a class object is made const, other than during initialization or destruction, any modification to any value is disallowed, even if it is done through a class method itself! Infact a const object can’t even call a normal method, it can only call a const method.

A const method is a method that uses this keyword, and in doing so it restricts any modification to the class object in it’s body. Aka const member function.

To use it,

#include<iostream>

class X{

private:

int x;

public:

X(int value): x{value}

{

}

X(): X{0} {

}

void setX(int value){

x=value;

}

int getX() const { //get consted

return x;

}

};

int main() {

const X x{};

x.setX(2); //error

std::cout<<x.getX()<<'\n';

}

We simply add const qualifier after function param list.

For adding const to a forward declared method,

class X{

private:

int x;

public:

int getX() const;

};

int X::getX() const {

return x; //ok

}

const methods can’t modify the data members, nor can they call methods that modify data members, i.e., const methods can only call const methods.

const methods can only return const reference to members, non-const methods can return const or non-const references.

For ex.:

class X{

private:

int x;

public:

int& getX() const {

return x; //error

}

const int& getX1() const {

return x; //ok

}

};

const methods can be overloaded based on the const qualifier,

int getX() {…}

int getX() const {…}

are treated differently.

The return type is still not considered for overload resolution so even it doesn’t matter if return type is const or not or even the same type, it’s upto the params and the const qualifier only.

* + 1. Static Data members

Just like normal identifiers, data members and function members in classes/structs can be made static.

And just like normal static identifiers, these have a static duration, i.e., lifetime from program start to program end.

However they can only be accessed throught the class object or class scope.

For ex.

class X{

public:

static int x; //static int x{}; would be an error

};

int X::x{}; //required or we can’t even assign a value to x.

int main() {

X a;

X b;

a.x=2; //ok

b.x; //returns 2

X::x; //returns 2

}

A static member is shared to all the objects an can be accessed as long as the class or it’s object is accessing it. They can be thought of as belonging to the class but in reality they are exactly like static global variables just scoped by the class.

Static class data members can only be declared inside the class, not defined. Because if they were allowed to be defined then they would be reset at every object instantiation. To define a static class data member we need to define them outside, much like forward declared class functions.

Syntax:

<data type> <class name>::<static class data member name>{…};

or use = instead of {…}, it’d be copy initialization in that case.

And just like forward declared class functions, if we have a header and cpp file, we define them in the cpp file. Static class data members also ignore access specifiers for initialization.

const and constexpr static class data member

If the data type is const integral type or const enum then it can be initialized with declaration.

For ex.;

class X{

private:

static const int a{1}; //ok

};

And any constexpr data member can be initialized with declaration.

class X{

private:

static constexpr std::array<int, 3> a{1,3}; //ok

};

constexpr and const require stack allocated types and not dynamically allocated types, they can still be used with them but the behavior isn’t well defined. That is

class X{

private:

static constexpr std::string a{1}; //ok but if we use a in code, it will not work

};

Inline Static class data member

From C++17, if the data type is inline, then regardless of constness it can be initialized with declaration.

class X{

private:

static inline int a{1}; //ok

};

* + 1. Static Class Functions

aka Static methods. Defined like normal.

For ex.:

class X{

private:

static const int a{1};

public:

static int getA(){

return a;

}

};

int main() {

X x{};

x.getA(); //returns 1

}

These methods don’t belong to any object specifically, and they don’t get \*this either, in getA(); return a is compiled to become X::a;

This is why static methods can’t access non-static members of a class.

When static methods are forward declared, the syntax requires static to be dropped for the definition.

For ex.:

class X{

public:

static int x;

static int getX();

};

int X::x{};

int X::getX() { //ok

return x;

}

A class that has only static members is called a pure static class or a monostate, since there is only 1 state and that is a global state.

There’s no such thing as a static ctor in C++, so we have to use odd tricks to initialize other static data members.

* + 1. Friend Functions

A friend function is a function that can access all members of a class despite not itself being a member of it. It ignores access specifiers. We can simply give a forward declaration of a function with the friend keyword as prefix inside the class to make the function a friend.

For ex.:

class X{

private:

int x;

public:

X(int value): x{value}{

}

friend int getX(X&);

};

int getX(X& x) {

return x.x;

}

int main() {

X a{1};

getX(a); //ok returns 1

}

The class just needs a forward declaration of the friend function.

Since the friend function isn’t a part of the class, it doesn’t have a \*this added to it at compile time automatically.

A class can have multiple friend functions and a friend function can belong to multiple classes.

For ex.:

class Y; //forward declaring a class

class X{

private:

int x

public:

X():x{0};

friend int add(X&,Y&); //we can use Y& here despite Y being not defined yet

};

class Y{

private:

int y;

public:

Y():y{2};

friend int add(X&,Y&);

};

int add(X& x, Y& y) {

return x.x+y.y; //works

}

A class can be forward declared just like a function. This is aka class prototype. Although a forward declared class can only be used by forward declared class methods or friend functions. In the case of forward declared class methods, the function definition must come after the class definition.

For ex.:

class Y; //class prototype

class X{

public:

void useY(Y);

};

class Y{ //actual definition of Y

public:

int a;

public:

Y(int value): a{value} {

}

};

void X::useY(Y y) {

}

int main() {

X x{};

Y y{1};

x.useY(y);

}

This forward declaration of class and stuff is only required when both classes are in the same cpp file. If we use the convention of separating classes into their header files and cpp files we wouldn’t need to do any of this, a simply #include “X.h” and #include “Y.h” would work.

A friend function can be directly defined inside the class instead of being forward declared and then defined later as well. Either way, a friend function is not a member of the class so the class members can never call a friend function,

Friend functions can call each other like normal functions.

class X{

private:

int x;

public:

X(int v1): x{v1} {

}

friend void yo();

friend void pp();

};

void pp(){

}

void yo() {

pp(); //works

}

int main() {

}

* + 1. Friend class

Instead of manually making each function a friend, C++ allows making an entire class a friend. Though this only applies to the member functions of the friend class and not the class itself.

For ex.:

class X{

private:

int x;

public:

X(): x{0}{

}

friend class Y; //doesn’t need forward declaration like friend functions

};

class Y{

private;

int y;

public:

Y(): y{0}{

}

int add(X& obj)

{

return obj.x + y; //can access \*this so Y& and X&’s private members since the class is a friend.

}

};

int main() {

X x{};

Y y{};

y.add(x); //ok returns 0

}

Works, but friend classes are simply a shorthand for making a bunch of functions friend functions, the class itself cannot access X’s members, so int y{x}; in Y would be an error.

If class A is a friend of class B, then this doesn’t make class B a friend of class A.

If class A is a friend of class B, and class B is a friend of class C, then this doesn’t make class A a friend of class C.

friend classes and functions harm encapsulation so they should be limited.

We can also make individual member functions a friend, i.e., friend member functions are also allowed.

For ex.:

class X; //forward declaration is necessary here for this example

class Y{

public:

void useX(X&); //can’t define it here since X isn’t defined yet.

};

class X{

private:

int x;

public:

X():x{1} {

}

friend Y::useX(X&); //friend member function, this is why we needed to define the class Y first, otherwise the compiler wouldn’t know if Y even has a useX method.

};

void Y::useX(X& x) { //can use X now

x.x; //returns 1

};

int main() {

X x{};

Y y{};

y.useX(x); //ok

}

* + 1. Anonymous Class Object

Just like we have Anonymous Objects for expressions, we can also make class objects Anonymous, i.e., give them expression scope and have them destroyed in the same statement.

For ex.:

class X{

public:

int x{3};

};

void useX(X& x) {

//do stuff with x

}

int main() {

useX(X{}); //works, here X{} creates an anonymous class object for X

X{}.x; //returns 3

X{10}.x; //returns 10, recall that without a ctor passing values to class/struct initializer passes it to its members

}

And just like normal Anonymous Objects, the object’s scope is limited to the expression itself.

* + 1. Nested Types

Since Classes and Structs are identical, even classes can have nested types, like nested classes / nested enums / nested structs.

* + 1. Struct vs Classes

It is recommended to use structs if there are only data members and not functions, because that is how they are in C. For only this reason a struct may not be called ill-formed if it doesn’t have a dtor, but a class will always be called ill-formed if it doesn’t have a dtor. This may lead to structs without dtors to be used around and they would not cleanup after being used.

Hence, for semantic reasons it is recommended to use classes where cleanup is desired.

As for technical difference between structs and classes, classes are private, structs are public by default and classes apply inheritance access specifiers to private by default whilst structs apply inheritance access specifiers to public by default. That’s it.

* + 1. std::reference\_wrapper

In some places, such as vectors, we may want to make a vector of T& instead of T, this helps us avoid copying the object and any change made to the T& in the vector would also affect the object.

By default, this is not possible, that is

#include<vector>

class X{…};

int main(){

std::vector<X&> vec{};

}

is an error, because X& isn’t an assignable type (recall that refs can’t be reassigned) and the vector requires each value of it to be able to be assigned and return value to implement the [ ] operator.

To do this, we can make use of std::reference\_wrapper in #include<functional>, this class takes any type T and acts like T& except it also allows assignment (non anonymous class type objects, because if they are anonymous class type objects then they’d not live longer than the expression and wouldn’t be a valid reference) and we can get a value using its .get() method.

For ex.:

#include<vector>

#include<functional>

class X{

private:

int x;

public:

X(int v1): x{v1} {

}

};

int main(){

std::vector<std::reference\_wrapper<X>> vec{}; //ok

vec.push\_back(X{2}); //error, as X{2} is an anonymous class object

X x{2};

vec.push\_back(x); //ok

X y{vec[0].get()}; //returns x when we use .get()

X z{43};

vec[0]=z; //works, vec[0] now holds a std::reference\_wrapper to z, which is like a reference to z

}

Other than for y, no copies were made of any object of X here.

A simpler example:

#include<iostream>

#include<functional>

class X{

public:

int x;

public:

X(int v1): x{v1} {

}

};

void yo(std::reference\_wrapper<X> xref)

{

std::cout<<(xref.get().x)<<'\n';

X x{2};

xref=x;

std::cout<<(xref.get().x)<<'\n';

}

int main(){

X x{3};

yo(x);

}

works.

Basically std::reference\_wrapper is simply a class that holds references to other objects and can change which object it references.

* 1. Encapsulation

aka information hiding. We hide stuff from the program that we deem unnecessary, this way we guarantee it won’t be affected by the program in any other way.

The access specifiers are a prime way of enforcing this.

Getters/Setters are a way of enabling this as well. We make methods that either get or set data members, this way we guarantee the change is controlled by the class itself. These methods are called access functions. Since the data members have a greater duration than the function body, getters can return by reference too, it’s recommended to return them as const references though.

* 1. Object Relationship

OOP paradigm focuses on objects and their relationships with other objects (in both the literal and figurative sense, i.e. objects as in types and objects as in the instantiations of those types).

* + 1. Object Composition

An object built of many smaller objects/parts, they ‘compose’ this object and their relationships with this object are considered in object composition.

For ex.: A car ‘has-a’ battery, a car ‘has-an’ engine and I ‘have-a’ pair of legs and so on.

Since classes and structs enable Type nesting, they are called composite types. The ‘object’ is the class/struct itself (their definition to be precise) and the ‘parts’ of this object are its members.

Object composition has 2 main subtypes

Composition

Aggregation

* + - 1. Composition

When an object and its part satisfy these relationships:

* The part is part of this object.
* One part can only belong to one object at a time.
* The part’s existence is managed by this object.
* The part’s not aware of this object, that is, it doesn’t directly depend on it.

For ex.: A class with some normal members.

A good rule to follow is the single-responsibility principle with classes/structs. That is, a single class should only try to represent a single object, like a car, or a human etc. but not try to mix them up in its definition, like a car with a human in it. Inheritance is a different thing in this context.

* + - 1. Aggregation

When an object and its part satisfy these relationships:

* The part is part of this object.
* One part can belong to >1 object at a time.
* The part’s existence is not managed by this object.
* The part’s not aware of this object, that is, it doesn’t directly depend on it.

For ex.: A house address can have a human in a house, but that human can change whilst the address remains the same and the house can change for a given human, so the human is a part of the house object yet neither’s existence depends on the other.

More specifically, these are dynamically allocated members of classes who are not initialized/destroyed with the class.

* + - 1. Association

A weaker type of relationship between a part and an object.

* The part is unrelated to the object.
* One part can belong to >1 object at a time.
* The part’s existence is not managed by this object.
* The part may or may not be aware of the object.

In other words, this is a ‘uses-a’ relationship, like a bank uses an ATM, the ATM can go to any bank and a bank can hold any ATM.

These are basically 2 classes that are a friend of each other, or their parts are friends of each other.

Reflexive Association

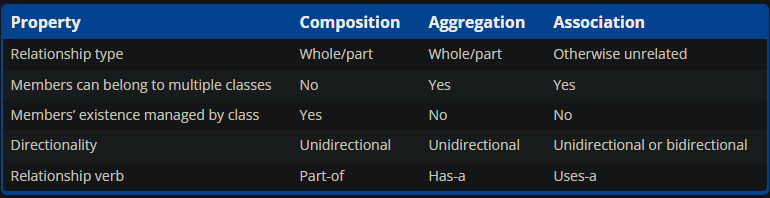
When an object has a relationship with itself, like a class method that accepts object of the same class (except \*this, copy ctor, and the like). For ex.: A ctor that accepts an object of the same class and initializes itself based on the values from the other object.

Indirect Association

An even weaker relationship, here an object has a relationship with its part but without directly having a relationship.

For ex.: A class may keep a list of IDs and then use any ID to get a class object with that ID, like a database primary key.

* + - 1. Relationship Table



* + - 1. Dependency

An even weaker relationship than association, a dependency is when one object invokes another object’s functionality to do something.

Like a function that uses a class object to do something.

The difference between association and dependency is that association has atleast some sort of link (like friend functions to the other class) but here the class is completely unaware of the function.

* + 1. Container Class

A container class is a class designed to hold multiple instances of another type. Like std::vector is a container class that has instances of T that we define. In C++, container classes should only contain instances of a single type of class, it’s not an error otherwise, but anti-pattern. This is why std::array and std::vector only hold a single type.

There are 2 types of container classes in C++, value containers, these are compositions that store copies of objects and reference containers, these are aggregations that store pointers and refs to objects.

* + - 1. Initializer List

An initializer list is a uniform initializer that has the form {<elem1>, <elem2>…}. These are automatically converted into an object of type std::initializer\_list by the compiler.

Meaning if we have a function that has a std::initializer\_list param, we can pass an initializer list to it as arg.

std::initializer\_list is in the #include<initializer\_list>

For our uses, it only accepts a single type, and behaves just like an array.

For ex.:

#include<initializer\_list>

void yo(std::initializer\_list<int> il) {

int sz{static\_cast<int>(il)}; //gets the size of il

for(int elem: il) {

//…gets each element in elem

}

}

int main() {

yo({1,2,5}); //ok

}

If we use initializer list in the ctor of a class then it will catch non-initializer list as well. That is,

#include<initializer\_list>

class X{

public:

X(std::initializer\_list<int> il)

…

}

};

int main() {

X x{1,2,3}; //ok, calls our initializer list ctor

X y{1}; //ok, still calls the same ctor

}

The fix is to use direct initialization in this case,

X z(2); //calls a single param ctor

X m({1,2,3}); //calls an initializer list ctor

This is why we use this to initialize types like vector with direct initializer, i.e.,

std::vector<int> a(5); //5-sized vector, all elem 0

std::vector<int> a(5,2); //5-sized vector, all elem 2

This is equivalent to

auto vec{std::vector<int>(5,2)};

except a copy doesn’t need to be made.

* 1. Inheritance

This is an ‘is-a’ relationship, like a wheel is a part of a car.

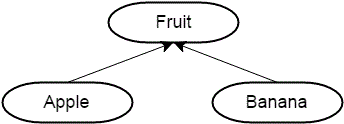
In programming, inheritance allows us to directly use attributes of the object in the part.

Like a wheel is a part of the object car, same for an engine, so both engine and wheel would know which car they belong to and can access the attributes of the car.

Both classes and structs support inheritance in C++.

In C++, inheritance provides the ‘is-a’ relationship, the class being inherited is called the parent class/base class/superclass and the class inheriting it is called a child class/derived class/subclass.

For ex.:



Here Fruit is the base class and Apple and Banana are derived classes.

The child class inherits all members of the parent class, though they are subject to access specifiers of the parent class.

We can define a parent class like normal, and a child class with this syntax.

class <child class name>: <access specifier> <parent class name> {<child class definition>};

For ex.

class X {

public:

int x;

X(): x{0} {

}

};

class CX : public X { //ok, now CX can use all members of X (subject to access specifiers)

public:

int y;

CX(): y{0} {

}

};

int main() {

CX c{};

c.x=2; //works

}

We can chain inheritance as well, so the class CX can be inherited too and it works exactly like here. The child of CX will also have all members of X, but it will also have all members of CX and then its own members.

* + 1. Order of Construction and Destruction

When we create an object of a child class, it’s just an object of the child class but the base classes are also instantiated to have their members in the child class instantiated.

That is, all the base classes have their ctors called.

For ex.:

#include<iostream>

class X {

public:

int x;

X(): x{0} {

std::cout<<”X”<<’\n’;

}

};

class CX : public X {

public:

int y;

CX(): y{0} {

std::cout<<”CX”<<’\n’;

}

};

int main() {

CX c{};

}

prints “X” then “CX”.

This is because the ctor that we invoke at object definition (so CX() here) delegates the ctor call to the base class (X() here) and that ctor delegates to its base class and so on until the most-base class is reached and then the ctor body of that class is invoked, then it returns and the ctor body of its child is invoked and so on until the most-child class is reached back again, so here X()’s body is executed first and CX()’s body is executed and that’s it but CX() is called first and then X() is called immediately before CX’s own body is executed.

If a ctor doesn’t explicitly define which ctor of the base class to delegate to, the default ctor of the immediate base class is invoked.

As for destructors, they have no overload, and can’t be called explicitly. In C++, the dtor chain is called in the reverse order of construction, so the most-child class is destructed first, then it’s base and so on until the most-base class.

* + 1. Initialization of derived classes

To initialize base class from a derived class we can explicitly call the base’s ctor. This is done using the Base keyword.

For ex.:

#include<iostream>

class X {

public:

int x;

X(): x{0} {

std::cout<<”X”<<’\n’;

}

X(int v1): x{v1}{

std::cout<<”X(int)”<<’\n’;

}

};

class CX : public X {

public:

int y;

CX(): Base{}, y{0} { //calls the default Base ctor

std::cout<<”CX”<<’\n’;

}

CX(int v1): Base{v1}, y{v1} { //calls the 1 param ctor of Base

std::cout<<”CX(int)”<<’\n’;

}

};

int main() {

CX c{}; //prints CX then X

CX c2{}; //prints CX(int) then X(int)

}

We can assign values to base members in the ctor of derived class as well, but that is just assignment, not initialization, C++ doesn’t allow us to initialize Base’s members in a derived class, so we can’t initialize x of X in CX’s member initializer list. This is to prevent double initialization, which can cause issue with references and pointers.

We can create a custom ctor call chain like this if we have multiple levels of inheritance.

* + 1. Inheritance and Access Specifier

Access Specifiers apply to 2 places when inheritance is concerned as well, firstly on the class members and secondly on the inheritance. In both cases, the same 3 access specifiers are used.

Access specifiers on the members

Private: The member is only accessible by the class members and its friends, not the derived classes or the callers outside.

Public: The member is accessible by all.

Protected: The member is accessibly by the class members, friends and the derived classes but not accessible by outside callers.

Access Specifiers on the Inheritance

Changes the access specifier of the member for the derived class. No matter which access specifier we apply on the inheritance, the private members of the base class will always be inaccessible to the derived class and outside the derived class.

Public: Makes no changes to the access specifiers of the members, private members remain inaccessible to the derived class and the outside.

Private: Makes the access specifiers of the members private, so now all non-private members of the base class are only accessible to the derived class.

Protected: Makes the access specifiers of the members protected, so all non-private members of the base class are accessible by the derived class and any class that will inherit this derived class.

For ex.:

class X{

private:

int x;

protected:

int y;

public:

int z;

X(): x{0},y{0},z{0} {

}

};

class Y: public X {

//y and z are accessible.

public:

Y() {

y=2; //ok

z=2; //ok

//x=2; //error, inaccessible

}

}; //any class that inherits Y won’t see x, but y and z will be accessible

class Z: private X {

//y and z are accessible

Z() {

//same as Y()

…

}

}; //any class that inherits Z won’t see x,y,z

int main() {

Y y{};

y.z; //ok, returns 2

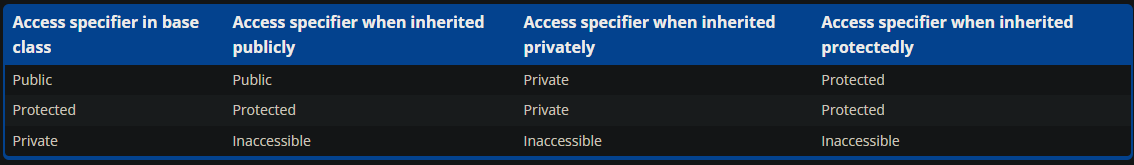
y.y; //error, inaccessible

Z z{};

z.z; //error

}

Here’s a table for it



By Default C++ inherits classes privately, so if we don’t specify the inheritance access specifier, it will be private by default.

For structs, the default is public.-

We can change the access specifier of an inherited member too, we use the using keyword for this.

For ex.:

class X{

protected:

int x;

public:

X(): x{0} {}

};

class Y: public X{

public:

using X::x; //X:: is defined later in this doc. Applies the public access specifier to x.

};

int main() {

Y y{};

y.x=2; //works

}

We can only change the access specifier of a member that is already visible to the derived class, so private members can’t have their access specifier changed like this.

This also allows us to hide functionality that is public in the base class but private in the derived class.

* + 1. Overriding members

If a derived class has any member with the same name as the parent class, it is accessed instead, just like shadowing.

For ex.:

#include<iostream>

class X{

public:

int x;

X(): x{0}{

}

};

class Y : public X{

public:

int x;

Y(): x{1} {

}

};

int main(){

Y y{};

std::cout<<y.x<<'\n';

}

prints 1.

This is because the executor looks for the member in the class, then it walks up the inheritance chain and gets the first member that we called. So if x isn’t defined in Y, it will look in Y’s parent, and then Y’s parent’s parent and so on until it finds it.

This allows us to redefine behavior.

We can access the Base’s members in derived class using <Base class name>::<member>.

For ex.:

class X{

public:

int x{0};

};

class Y: public X{

void yo() {

X::x=2; //ok

}

};

Outside the derived class, we can use <obj>.<base class name>::<members> to access accessible members using the object of the derived class. This is equivalent to upcasting the object.

That is,

class X{

public:

int x{2};

};

class Y: public X{

};

int main() {

Y y{};

y.X::x=3; //ok

y.x; //also ok, returns 3

//Alternatively

static\_cast<X&>(y).x=4; //ok

}

This is also to say, when a class object is instantiated, it has all the members of that class, and its parent classes, given that the access specifiers allows those members of the parent classes to be inherited down till the class whose object is being instantiated. The inheritable members theirselves are part of the deriving class but they can be hidden too, still they exist and if access specifiers allow, can be accessed. The inheritance access specifier of the base class defines if a base class can be accessed through the child class’ object.

That is,

#include <iostream>

class X{

public:

int x{2};

int\* p;

X(): p{new int{2}} {}

~X() {delete p;}

};

class Y: public X {

public:

int x{3};

int\* p;

Y(): p{new int{3}} {}

~Y() {delete p;}

};

int main() {

Y y{};

std::cout<<y.x<<'\n';

std::cout<<\*(y.p)<<'\n';

std::cout<<y.X::x<<'\n';

std::cout<<\*(y.X::p)<<'\n';

}

prints 3 then 3 then 2 then 2.

Here, if X was inherited as

class Y: private X {…}

then we wouldn’t be able to access y.X in main().

Friend functions aren’t part of the class itself, so we can’t call the friend functions from a derived class.

* + 1. Delete inherited methods

Methods in classes can be marked with this keyword to prohibit usage. This applies to methods in derived classes too.

For ex.:

class X{

public:

void yo() {}

};

class Y: public X{

void yo() = delete;

};

Now no object of y can call yo().

We can still upcast Y’s object or access the base class object through y and that would still work.

* + 1. Multiple Inheritance

A single class can inherit multiple classes in C++. And multiple classes can inherit a single class too.

For ex.:

class X{…};

class Y{…};

class Z: public X, public Y {

…

};

works just like expected.

Mixins

A mixin is a class that is intended to be inherited and not instantiated on its own. They generally don’t use virtual functions and are templatized (uses templates).

A derived class that uses a templatized mixin with its own type as the template parameter presents an inheritance called a Curiously Recurring Template Pattern (CRTP).

For ex.:

#include<iostream>

template<typename T>

class MixinA {

public:

void yo() {

//we can access members of T

static\_cast<T\*>(this)->x=3; //ok, casts \*this which is a MixinA& to T\* which has access to its child class members.

}

};

class X: public MixinA<X> {

public:

int x;

X(): x{0} {}

};

int main() {

X x{};

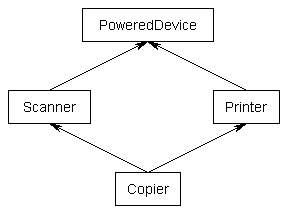
x.yo();

std::cout<<x.x; //returns 3

}

As we can see this is a troublesome pattern as the parent is modifying the child.

Another problem with multiple inheritance is the diamond inheritance problem



If we have an inheritance hierarchy like so, then if there are ambiguous (same named) methods in multiple parent classes then we won’t know which class’ methods are being called through the most-child class’ object.

This is why multiple inheritance should be avoided.

* + 1. Inheritance with References and Pointers

We know that we can set reference and pointer to a class and it works as expected, be it derived or not. But in C++ we can also use the pointer/reference of the base type to hold pointer/reference of the derived class’ object.

For ex.:

class Z {

public:

int z;

Z(): z{0} {}

};

class X: public Z{

public:

int x;

X(): x{0} {}

};

class Y : public X{

public:

int x;

Y(): x{0} {}

};

int main() {

Y y{};

Y& yr{y}; //ok

Y\* yp{&y}; //ok

yp->x=3; //ok

yr.z = 4; //ok

X& xr{y}; //also ok

X\* xp{&y}; //also ok

xr.x; //returns 0

xp->z; //works, returns 4

}

Whilst a Base class’ ref/pointer can point to a derived class’ object, the values it sees are for the instantiation of its own class and any class it is deriving.   
In other words, the object of Y has instantiation for X which has an instantiation of Z, so we can get the object of X from Y and it will have an instantiation of Z, but not of Y because Y is a child class.

This is an alternate way of accessing a base class, the other ways are either upcasting or just accessing a base class using scope resolution operator on the object of Y. But these ways are a slight bit different from pointer/reference of class objects as we see when virtual functions are concerned.

This behavior is known as covariance.

Covariants are supported in C++ but contravariants are trickier. Covariance is when a pointer/reference type holds a derived class’ object. Contravariance is when a pointer/reference type holds a more base class’ object.

For ex.:

#include<iostream>

class X{

public:

void yo() {

std::cout<<”Xyo”<<’\n’;

}

};

class Y: public X {

public:

void yo() {

std::cout<<”Yyo”<<’\n’;

}

};

int main() {

Y y{};

y.yo(); //prints “Yyo”

X& xr{y};

xr.yo(); //prints “Xyo”, this is covariance

X x{};

Y& yr{x}; //error, but using some trick if we achieve this then that’s contravariance

}

* + 1. Virtual Function

A special type of class function that when called resolves to the most-derived version. We simply use the virtual keyword on a function’s declaration to make it virtual.

For ex.:

#include<iostream>

class X {

public:

int x;

X(): x{0} {}

virtual void yo() {

std::cout<<”Xyo”<<’\n’;

}

virtual ~X() = default; //virtual destructor required

};

class Y : public X{

public:

int y;

Y(): y{0} {}

virtual void yo() {

std::cout<<”Yyo”<<’\n’;

}

};

int main() {

Y y{};

Y& yr{y};

yr.yo(); //prints “Yyo”

X& xr{y};

xr.yo(); //requires call syntax of X::yo() not Y::yo(), still prints “Yyo”

}

Even though the reference xr is of type X&, it accesses .yo() in a more-derived class for the object y, that is Y, so it access Y::yo(). This is in contrast to the default behavior where yo() wouldn’t be visible to a reference of X& if it was a normal function.

Virtual functions enable a capability known as polymorphism, that is, accessing an identifier changes behavior based on some context, here it is the object y.

A derived function is considered a match/override for a virtual function if it has the same name, params, constness and return type.

For ex.:

#include<iostream>

class X {

public:

int x;

X(): x{0} {}

virtual void yo() {

std::cout<<"Xyo"<<'\n';

}

virtual ~X() = default; //virtual destructor required

};

class Z: public X{

public:

virtual void yo() {

std::cout<<"Zyo"<<'\n';

}

};

class Y : public Z{

public:

int y;

Y(): y{0} {}

virtual void yo() const {

std::cout<<"Yyo"<<'\n';

}

};

int main() {

Y y{};

Y& yr{y};

yr.yo(); //prints “Yyo”

X& xr{y};

xr.yo(); //requires syntax of X::yo(), prints “Zyo”, would have printed “Xyo” if Z didn’t exist.

}

Notice this example is almost the same as the one before, but the only difference is constness of yo() in Y, because of which a different function is called through xr.

This is because the virtual function yo() in Z is not ‘overridden’ in Y, override occurs when a derived class has an exact same signature virtual function. So when X&’s object tries to call yo(), it looks for a definition that exactly matches it’s own yo() in the next derived class, if it doesn’t find then it stops and simply accesses its own class’ yo(). Here the executor when invoked on object of X& looks for X::yo()’s override in Z, it finds it, then it looks for Z::yo()’s override in Y, it doesn’t find one so it runs Z::yo() and doesn’t bother going down the inheritance chain (if any).

One more thing to keep in mind is that when we call yo() on X&, the compiler requires X::yo()’s syntax to be used and not Y::yo() or Z::yo() because it doesn’t know which function will be called just yet. That is also to say, polymorphism is a runtime property.

Virtual functions only work when called through a pointer or reference, if a normal object or an upcasted object tries to call a virtual function, it simply looks in it’s own class for the functions (be they inherited, overridden or directly defined in that class).

For ex.:

class X{

public:

virtual void yo() {

}

};

class Y: public X{

public:

virtual void yo(){}

};

int main() {

Y y{};

y.X::yo(); //will call X::yo() no matter what.

}

If a base class’ function is marked as virtual, all derived classes that override this function are implicitly marked as virtual if they don’t specify virtual keyword.   
This doesn’t work the other way around when a base class doesn’t mark its function as virtual but a derived class does, a more derived class can override the base class’ function and even mark it as virtual but when base class’ pointer/reference calls that function, they don’t look for an override.

#include<iostream>

class X {

public:

int x;

X(): x{0} {}

virtual void yo() {

std::cout<<"Xyo"<<'\n';

}

virtual ~X() = default; //virtual destructor required

};

class Y : public X{

public:

int y;

Y(): y{0} {}

void yo() {

std::cout<<"Yyo"<<'\n';

}

};

int main() {

Y y{};

Y& yr{y};

yr.yo(); //prints “Yyo”

X& xr{y};

xr.yo(); //still prints “Yyo”

}

But

class X {

public:

int x;

X(): x{0} {}

void yo() {

std::cout<<"Xyo"<<'\n';

}

virtual ~X() = default; //virtual destructor required

};

class Y : public X{

public:

int y;

Y(): y{0} {}

virtual void yo() {

std::cout<<"Yyo"<<'\n';

}

};

int main() {

Y y{};

Y& yr{y};

yr.yo(); //prints “Yyo”

X& xr{y};

xr.yo(); //prints “Xyo”

}

Virtual functions that override a function need to have the same return type.

Virtual functions can’t be called from ctors or dtors. This is because recall that when a ctor is executed, the most-base class’ ctor is invoked first and then it’s children, but if any ctor calls a virtual function then it will look for a definition in its derived classes and since the derived classes aren’t instantiated, this will lead to an error. Same for a dtor because it will look for an instantiation in destructed classes.

To define a virtual function body outside the class, we do it just like normal fucntions.

For ex.:

class X{

public:

virtual void yo();

};

void X::yo() {}

int main() {

X x{}; //error, can’t instantiate an abstract base class

x.yo(); //ok

}

* + 1. Override and Final

These are not keywords as they only have meaning in certain contexts, that’s why called specifiers instead.

Override enforces override for a virtual function, that is, if a virtual function is overriding a function from the base class, using this specifier ensures that it is doing so, and if it isn’t then that is a compile-time error. This avoids situations where we intended to override a virtual function but failed due to signature not being the same.

For ex.:

class X{

public:

virtual void yo() {}

};

class Y: public X {

public:

virtual void yo() override {} //ok

//virtual void yo() const override {} //error as yo() doesn’t override base class’ yo()

};

Using override implicitly adds virtual to a function so we can omit virtual keyword and only leave override if a function is overriding a base class’ function. There is no performance penalty with using override (apart from the implicit virtual keyword) so it is recommended to be used when overriding is intended.

Final specifier on the other hand prohibits overriding or inheritance.

Applied the same way as the override specifier, and can also be applied to a class to prevent inheritance of it.

For ex.:

class X{

public:

virtual void yo() {}

};

class Y: public X {

public:

void yo() override final {}

};

class Z final: public Y{

public:

void yo() override {} //error

};

class K: public Z {…}; //error

Z::yo() is an error as it is trying to override a function that has a final specifier. It doesn’t need to have the override specifier itself as it’d be an error anyway, but it is good practice to have it anyway.

Similarly, K can’t inherit Z because Z has been marked final. Final classes can’t be inherited.

Covariant return types

When an overriding function returns a covariant (a more derived class pointer or reference) but the base class doesn’t then that’s not an error as C++ knows that’s covariance and that works as-is.

For ex.:

#include<iostream>

class X{

public:

virtual X\* yo() {

std::cout<<"Xyo"<<'\n';

return this;

}

void print() {

std::cout<<"Xp"<<'\n';

}

};

class Y: public X {

public:

virtual Y\* yo() { //works

std::cout<<"Yyo"<<'\n';

return this;

}

void print() {

std::cout<<"Yp"<<'\n';

}

};

int main() {

Y y{};

X& xr{y};

y.yo()->print(); //prints “Yyo" then prints “Yp" as expected

xr.yo()->print(); //prints “Xyo" then prints “Xp"

}

xr.yo() is still inside X but the function yo() is a virtual function so the executor runs the Y::yo(), however passes X& instead of Y& to Y::yo()’s this implicitly. Then Y::yo() simply returns X\* in a Y\* (called covariance) and when print() is called on this Y\* it finds print() of X in the object so it calls that.

* + 1. Virtual destructor

Normal destructors are called when the class’ object is going out of lifetime, this is ok if the class doesn’t have any virtual functions and doesn’t allow inheritance but when a class does have virtual functions and allows inheritance, then it means that the class’ functions may be overridden in an object (pointer or reference),

For ex.:

#include<iostream>

class X{ //no final means class can be inherited

public:

virtual void yo() { ///virtual function, means it can be overridden

}

~X() { //normal destructor

std::cout<<”Xd”<<’\n’;

}

};

class Y: public X{

public:

void yo() override {

}

~Y() {

std::cout<<”Yd”<<’\n’;

}

};

int main() {

Y\* y{new Y()};

X\* xp{&y};

delete xp; //ok

xp=nullptr;

}

prints “Xd”. But this is a problem as we asked to deallocate the whole object and it only deallocated X from y, meaning now object y is in a state where it’s X members are uninitialized and if we call an object in y that depends on an object of X, then that would lead to undefined behavior. So the optimal behavior is to have the whole of y destructed.

To do so we make the destructor virtual, when the virtual destructor of a base class is called, all the destructors of its child classes and even the parent ones.

For ex.:

#include <iostream>

class X {

public:

virtual void yo() {

}

virtual ~X() { // virtual destructor

std::cout << "Xd" << '\n';

}

};

class Y : public X {

public:

void yo() override {}

virtual ~Y() { std::cout << "Yd" << '\n'; }

};

class Z : public Y {

public:

void yo() override {}

virtual ~Z() { std::cout << "Zd" << '\n'; }

};

int main() {

Z\* z{new Z()};

Y\* yp{z};

delete yp; // ok

yp = nullptr;

}

prints “Zd” then “Yd” then “Xd”.

Note that we did Z\* and assigned it to Y\*, to call delete the object must be fully on heap, so if z was of type Z and we assigned it’s address to Y\*, then whilst it would work as normal, calling delete explicitly would be an error as it would assume the Z part of the object Y\* was a pointer too, that it was on heap, but it’d be actually on the heap.

* + 1. Access Specifier on a virtual dtor and functions

Virtual functions can be overridden even with private access specifier.

For ex.:

#include <iostream>

class X {

public:

virtual void yo() {

std::cout<<"Xyo"<<'\n';

}

};

class Y : public X {

private:

void yo() override {

std::cout<<"Yyo"<<'\n';

}

};

int main() {

Y y{};

X& xr{y};

xr.yo(); //prints “Yyo”

}

This is to say, virtual functions don’t care about access specifiers when being resolved at runtime. This is true even when the base class has a private function and it’s being overridden in a derived class.

For ex.:

#include <iostream>

class X {

private:

virtual void yo() {

std::cout<<"Xyo"<<'\n';

}

};

class Y : public X {

public:

void yo(int) override { //error

std::cout<<"Yyo"<<'\n';

}

};

int main() {

Y y{};

y.yo(2);

}

It is an error because yo()’s signature doesn’t match yo() from X even though yo() from X is inaccessible to Y. If we fix the signature to match then y.yo() would work and print “Yyo”.

This behavior is slightly different with destructors, firstly, delete can only be called on a pointer of a PDT if the destructor is accessible to the caller,

For ex.:

class X{

private:

~X() {}

};

int main() {

X\* x{};

delete x; //error

} //destructor of x is called implicitly here thanks to RAII

so if a base class has a private dtor (virtual or not), and we call delete on its pointer, then it wouldn’t work and we have prevented dtor from being called for the entire inheritance chain of the object. But it also means the base class’ pointer can’t have a dynamically allocated base class object (as it can’t deallocate explicitly). This also means we can’t use the base class in a unique pointer. Lastly, for some reason this also means base class can’t be allocated on the stack.

All these reasons mean dtors should be virtual when public, but non-virtual when private or protected.

* + 1. Virtual assignment

A class’ assignment can be made virtual too.

* + 1. Binding

During compilation functions are given a memory to reside and its address is stored on the stack, then at runtime, the executor goes to the address and executes the function. The process of associating identifiers with their memory addresses is called binding. There’s variable binding and function binding.

* + - 1. Early Binding

Normal function calls can be resolved at compile itself.

For ex.:

void yo() {}

int main() {

yo(); //early binding

}

This call is called early binding / static binding because the compiler and linker can replace the call with the function’s address at compile time itself then at runtime the executor simply needs to jump to the address to continue execution.

* + - 1. Late Binding

Aka dynamic binding. Contrary to the early binding, here the compiler and linker don’t have an address for the function and the call isn’t resolved until runtime.

At runtime the function is dynamically searched for with signature.

C++ doesn’t support late binding, but virtual functions are as close to late binding as C++ allows. Their calls are resolved at runtime but not at the call-site, instead at some prior point of the program’s lifecycle.

In this context, C++ does support Late Binding.

Function pointers are also an example of late binding as the compiler can’t perform early binding due to pointer indirection.

For ex.:

void yo() {}

int main() {

void (\*p)() {yo};

p(); //address is resolved at runtime

}

The virtual table

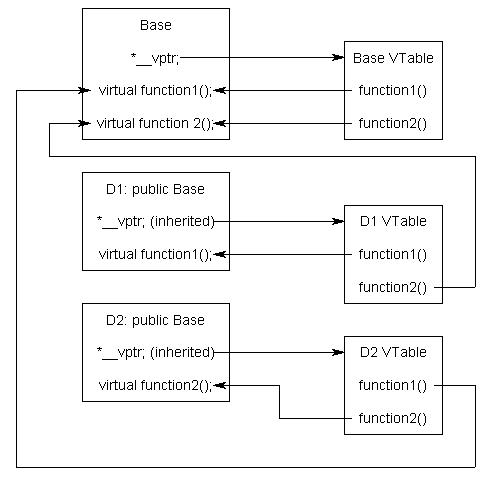
aka vtable or virtual method table or virtual function table.

C++ implements late binding on virtual functions using this lookup table, it manages this table at runtime and when a virtual function’s call has to be resolved, refers to this table.

The way C++ does it is like so, every class is assigned a vtable, this is a static array with function pointers in it, each function pointer points to a virtual function of the class. This vtable is a static array but it is defined outside the class in a different area (probably the heap), inside the class the compiler implicitly adds a pointer to this vtable.

Now when the class is derived, this vtable pointer is inherited as well and the derived class doesn’t create its own vtable pointer if it inherits it from the base. It still creates its own vtable though and sets the vtable pointer to it. And just like the base class vtable pointer, it adds virtual function pointers for the inherited virtual functions and also its own virtual functions’ pointer to the vtable, and if a virtual function signature of derived class matches a virtual function of the base in this table, it overwrites the address to point to its own, i.e., point to derived class’ virtual function. This is called virtual function overriding.

That is,



The vtable pointer is created for appropriate classes at compile time itself (so the class takes 1 pointer worth extra space), but the vtable’s address is only resolved at runtime, and the vtables are filled out during object creation.

One important thing to know is that whilst each class basically has its own vtable and vtable pointer, and that vtable is merged from the most-base to the most-derived class, the vpointer is modified as well.

For ex.:

#include<iostream>

class X{

public:

virtual void yo(){

std::cout<<”Xyo”<<’\n’;

}

};

class Y: public X {

public:

void yo() override {

std::cout<<”Yyo”<<’\n’;

}

};

int main() {

Y y{}; //creates a y object

X& x{y};

x.yo(); //prints “Yyo”

}

When a pointer/reference object of a class calls a virtual function, the function’s signature in the type of the object is looked for in the vtable of that class through its vtable pointer.

Here when object y is created, it resolves the vtable pointer in Y (inherited from X) and overwrites the yo()’s address in the vtable with the one from Y with the yo() inherited from X. As we can see a reference of type X accesses an object of type Y in it, and its vpointer should point to X’s vtable which has yo() of X, but it doesn’t. This is because during object instantiation vpointer is modified as well, the vtable address pointed to by the most-derived class is copied to the vtable pointers of all base classes.

So in this example, Y’s ctor calls X’s ctor, X’s ctor instantiates vtable pointer of X to point to vtable of X, then the ctor returns and we are back in Y, now Y’s ctor runs and instantiates vtable pointer of Y to point to vtable of Y, and it also modifies the vtable pointer of X to point to vtable of Y.

So, when X& calls yo(), it looks in the vtable pointed to by its vpointer, which is still of Y and hence Y::yo() is called.

This indirection is the reason why virtual function calls are slower than normal function calls.

* + 1. Abstract Base Class

A class with >=1 pure virtual functions is called an abstract base class, such classes can’t be instantiated.

A pure virtual function, aka abstract function, is a virtual function, that leaves its implementation/definition upto the deriving class.

For ex.:

class X{

public:

virtual void yo() const = 0; // = 0 makes a virtual function a PVC

};

class Y: public X {

public:

void yo() const override {}

};

int main() {

X x{}; //error, can’t instantiate an abstract base class

Y y{};

y.yo(); //ok

}

If a class derives an abstract base class and doesn’t provide definition for a PVC, inherited or not, then it is also an abstract base class.

PVCs are virtual functions too, so they follow the same binding, overriding and behaviors as normal virtual functions.

PVCs can be defined, but they can only be called later on and don’t make the class non-abstract base class.

For ex.:

class X{

public:

virtual void yo() const = 0; //PVC

};

void X::yo() const {}

class Y: public X { //X is still an abstract base class

public:

void yo() const override { //override is necessary to make Y non-abstract

X::yo(); //ok, since the definition actually exists

}

};

int main() {

Y y{};

y.yo(); //ok

}

Abstract Base classes have vtables and vtable pointers too, but the address of PVCs is stored as either null pointer or a generic function, sometimes called a \_\_purecall function, which just generates an error on being called directly.

Interface

In C++, abstract base classes that have no data members and only have PVCs are called Interfaces.

* + 1. Virtual Base Class

When inheriting a class, we can add the virtual keyword to make the inheritance virtual and the base class is then called a virtual base class.

A virtual base class shares its object and is recreated.

For ex.:

#include <iostream>

class X{

public:

X() {

std::cout<<"X"<<'\n';

}

};

class Y : X{

public:

Y() {

std::cout<<"Y"<<'\n';

}

};

class Z: X{

public:

Z() {

std::cout<<"Z"<<'\n';

}

};

class K : Z, Y{

public:

K() {

std::cout<<"K"<<'\n';

}

};

int main() {

K k{};

}

prints X then Z then X then Y then K as expected with normal order of construction.

This is the famous diamond inheritance problem. As we can see multiple copies of X are created, meaning if anything modifies k.X then it is not clear which X would be modified.

This is where virtual base classes help,

#include <iostream>

class X{

public:

X() {

std::cout<<"X"<<'\n';

}

};

class Y : virtual public X{ //the public access specifier is needed for K to be able to call X

public:

Y() {

std::cout<<"Y"<<'\n';

}

};

class Z: virtual public X{ //same as above comment

public:

Z() {

std::cout<<"Z"<<'\n';

}

};

class K : Z, Y{

public:

K(): X{} {

std::cout<<"K"<<'\n';

}

};

int main() {

K k{};

}

prints X then Z then Y then K.

Virtual inheritance requires the most-derived class to instantiate the base classes, and in this case alone can a class call a indirect parent class (like K can call X). Then this instance is shared to the non-virtual parent classes.

That is also to say, virtual base classes are constructed before non-virtual ones, so the non-virtual classes get the virtual class instances after they are created.

The way this works is through vtable, and every class inheriting a virtual class will have a virtual table regardless of having virtual functions. And with the help of some virtual table logic they resolve which class to point to. Also meaning that these classes will have a pointer added to their size.

* + 1. Object slicing

When a base class pointer/reference refers to a derived class’ object, then through inheritance/overriding etc. we can access overridden methods.

For ex.:

#include <iostream>

class X {

public:

virtual int yo() { return 2; }

};

class Y : public X {

public:

int yo() override { return 2; }

};

int main() {

Y y{};

std::cout << y.yo() << '\n';

X\* x{&y};

std::cout << x->yo() << '\n';

}

prints 3 both times as expected.

But when the whole object is ‘upcasted’, that is, when a derived child’s object is assigned to a base class then the child object’s classes are removed from the object until the base class and base class’ parents are the only ones left. This is called Object slicing.

For ex.:

#include <iostream>

class X {

public:

virtual int yo() { return 2; }

};

class Y : public X {

public:

int yo() override { return 2; }

};

int main() {

Y y{};

std::cout << y.yo() << '\n';

X x{y};

std::cout << x.yo() << '\n';

}

prints “Yyo” for y.yo() but “Xyo” for x.yo() as the Y in y is sliced off when being assigned and copied into X. Yes, this is an assignment copy operation.

This doesn’t happen with references and pointers, which is why it is recommended to pass them instead of raw value to functions, because they may slice the object without any issues and that may be unwanted if we wanted to rely on overridden methods.

But slicing does occur even with references (not pointers).

For ex.:

#include <iostream>

class X {

public:

int x;

X(int v1):x{v1}{ }

};

class Y : public X {

public:

int y;

Y(int v1): X{v1}, y{v1} {}

};

int main() {

Y y1{2};

Y y2{3};

X& x{y1};

std::cout << y1.x << '\n';

std::cout << y1.y << '\n';

x= y2;

std::cout << y1.x << '\n';

std::cout << y1.y << '\n';

}

Prints 2 then 2 then 3 then 2. Lo and behold, y1 is a FrankenObject!

This only happens when object type is a reference of a class object and not when it’s a pointer because its not reassignment of reference x, no, this is copy assignment of y2 into y1. So y2 is being copied into y1 because of the assignment operator.

But that’s only the half of it, since x is of type X& the implicit copy assignment overload of X is invoked, which is a non-virtual function by default and it gets y2 as const X& and performs memberwise copy, but it only knows the members of X in the implicit copy assignment operator overload so it only copies the members of X from y2 to y1.

The result is a kind of object slicing and y2’s base class X members are only copied into y1 whilst the Y class members are left as-is as they are from a more derived class than X.

This is why y1 retains its original Y members even though y2 was copied into it, all thanks to the X&.

* + 1. Downcasting

We know we can directly assign derived class object into a base class object, pointer or reference, called upcasting. But we can also return this upcasted object back into a derived class object, this is called downcasting and in C++ we use dynamic\_cast to do so.

For ex.:

#include <iostream>

class X {

public:

virtual void yo(){}

};

class Y : public X {

public:

void yo()override {};

};

int main() {

Y\* y{new Y{}};

X\* x{y};

Y\* dY{dynamic\_cast<Y\*>(x)};

}

works.

dynamic\_cast has some strict requirements, firstly the base class must have a vtable, so it must have a virtual function in it, secondly the derived class must inherit the base class as public (private and protected prohibit it from seeing the members).

It can still fail, if it fails it will return nullptr so we must make a check for nullptr on a dynamic\_cast’s result.

dynamic\_cast performs some checks at runtime to check if the conversion will succeed, this incurs a performance penalty.

We can also use static\_cast to forgo any runtime checking but the result is, that if the pointer isn’t a valid pointer to a derived class already then it will simply return derived class and we would get undefined behavior if we access derived class objects.

dynamic\_cast can also be used with refs similarly.

C++ has a concept called RTTI, Run-time Type Information, this provides information about an object’s type at runtime but incurs a huge memory and performance cost so RTTI can be disabled as an optimization. However, doing so affects dynamic\_cast as it relies on RTTI to function correctly.

* + 1. Class Template

Works same way as for structs. A class template is simply a definition for a template type that may be used later in the code and once it is the compiler has to create that specific definition at compile time, this is called instantiation for a template type. With classes, it’s like a type(template) for a type(class).

1. Chrono

C++’s time library. Has many useful methods for working with Time.

Use with #include<chrono>

* 1. Simple benchmark code

#include <chrono> // for std::chrono functions

class Timer

{

private:

// Type aliases to make accessing nested type easier

using Clock = std::chrono::steady\_clock;

using Second = std::chrono::duration<double, std::ratio<1> >;

std::chrono::time\_point<Clock> m\_beg { Clock::now() };

public:

void reset()

{

m\_beg = Clock::now();

}

double elapsed() const

{

return std::chrono::duration\_cast<Second>(Clock::now() - m\_beg).count();

}

};

and to use it,

simply create Timer t{};

and call t.elapsed(); after the piece of code whose time taken needs to be measured.

A good talk on benchmarking and performance (in general and not just C++ specific): <https://www.youtube.com/watch?v=r-TLSBdHe1A>

1. Operators

An operation is a calculation that involves zero or more values and produces a value from the calculation. An operand is a value and operators define the operation to be performed.

For ex.:

2+5

Here 2 and 5 are operands, and + is an operator. The operation here is addition and the result will be 7.

In C++ we have various operators such as +,-,=,<<,>> etc.

The no. of operands an operator accepts is known as its arity.

When multiple operators are present in a statement, C++ follows PEMDAS (Parenthesis, Exponents, Multiply …)

Not all operators return a value.

= and << are both operators as well. So what is their return type ? They don’t have a definite return value but return their left operands in C++.

x=5;

Here the operator = returns the variable x.

This allows for operator chaining, for ex.

int x;

int y;

x = y =5;

Here it is inferred as x= (y=5) and first 5 is copied to y then y is copied to x as y=5 returns y.

Similarly,

cout<<”yo”<<” aaa”;

is actually inferred as (cout<<”yo”)<<” aaa”

so first “yo” is sent to the output buffer then cout is returned and lastly “ aaa” is passed to cout.

* 1. sizeof

This operator takes 1 operand which is the name of an identifierand returns the size of the type in the memory.

For ex.:

sizeof(int);

will return 4 or 8 on 64bit machines.

It can also take variables, so

int a;

sizeof(a);

works and returns 4 or 8.

* 1. typeid

This operator takes an expression and returns the type information such as name.

For ex.:

#include<iostream>

#include<typeinfo>

int main(){

std::cout<<typeid(2+2).name()<<’\n’;

}

prints “signed int”.

However this operator doesn’t follow functions, so

void yo() {

std::cout<<”yo”<<’\n’;

}

int main() {

std::cout<<typeid(yo()).name()<<’\n’;

}

prints “void” and nothing else, as typeid here gets the return type of the function but doesn’t execute it.

typeid’s implementation is not standardized so it may return different type info on different compilers. For ex.: On GCC, it could return PKc for auto s{“yo”}; where PKc means pointer to const char. This is Itanium C++ ABI naming and names are mangled. We can get a demangled general name using cxxabi.h,

#include <typeinfo>

#include <cxxabi.h>

#include <iostream>

int main()

{

const char \*str = "Hello, world!";

std::cout << typeid(str).name() << std::endl; // prints "PKc"

int status;

char \*demangled = abi::\_\_cxa\_demangle(typeid(str).name(), nullptr, nullptr, &status);

if (status == 0) {

std::cout << demangled << std::endl; // prints "char const\*"

std::free(demangled);

}

}

[ChatGPT]

* 1. Precedence And Associativity

Operators in HLLs have a precedence level that defines which operator should be solved first in an expression.

But 2 operators could have the same precedence, such as \* and /.

In this case another factor is considered by the compiler, known as the associativity, which is defined for each precedence level and tells the compiler whether the expression should be solved from Left-To-Right or Right-To-Left.

For ex.:

3\*2/4

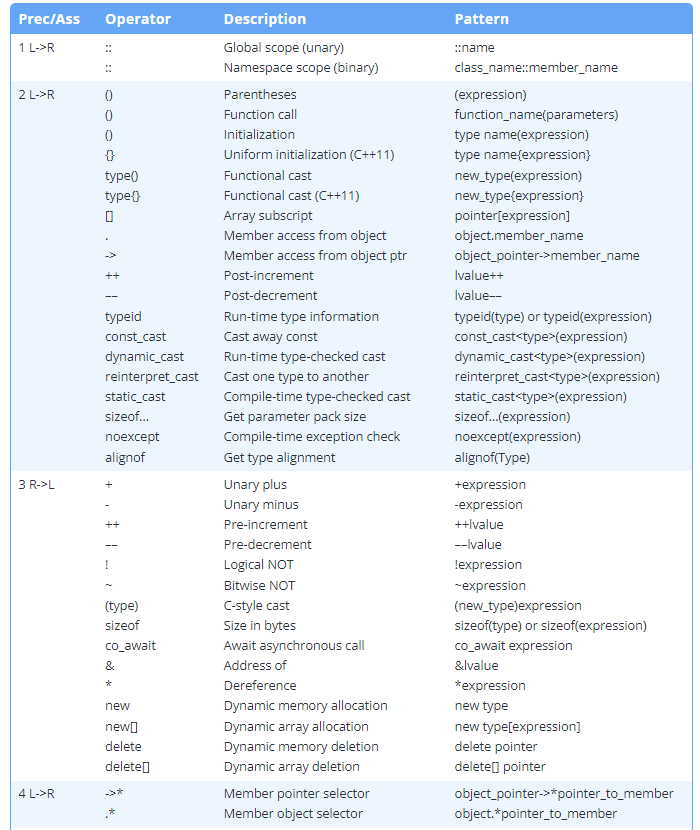
\* and / both have precedence level 5, and the associativity for level 5 is LTR so the expr is solved like so

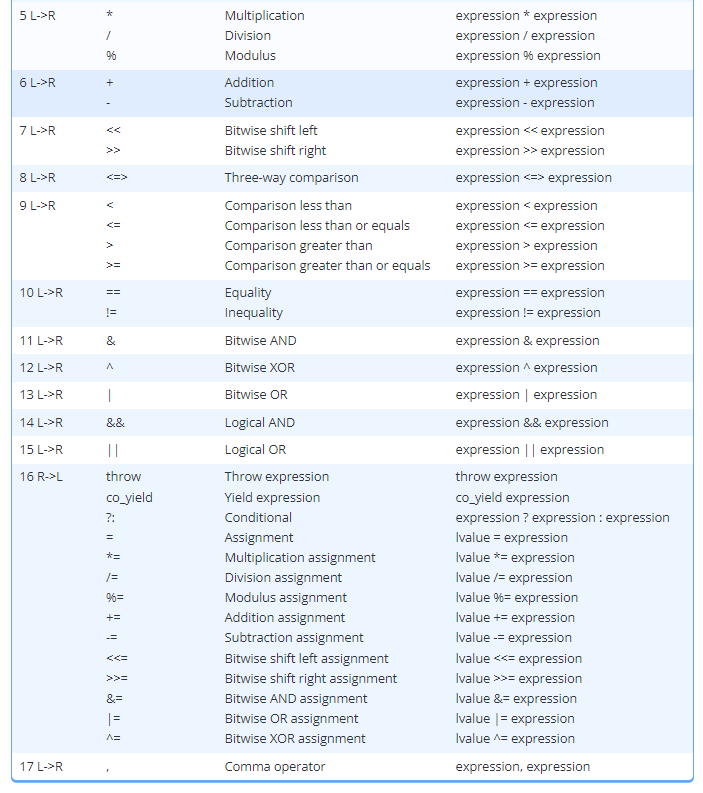
3\*2

then

6/4

* 1. Precedence And Associativity Table for C++





The lower the precedence, the earlier that operator is solved.

Parentheses have PL 2, making them very useful to both denote the right subexpression to the reader and also to tell the compiler what to do first.

For ex.:

x = (y \*= z);

However, the PL of variables, function calls etc. is upto the compiler.

For ex.:

#include <iostream>

int getValue()

{

std::cout << "Enter an integer: ";

int x{};

std::cin >> x;

return x;

}

int main()

{

std::cout << getValue() + (getValue() \* getValue()); // a + (b \* c)

return 0;

}

In this code if we enter 1 then 2 then 3 we might assume the expr. becomes 1+(2\*3) = 6 but it is entirely possible 2+(1\*3) or some other combination is made.

This is why we must make sure the expressions don’t get affected by order of evaluation of variables and calls.

For an op. the standard explicitly states that the left operand must be evaluated first and then the right one and then the actual operation.

* 1. Increment and Decrement Op.

++ and --,

x++ means copy x then increment the copy and return the incremented copy. ++x means increment x directly and return it.

Same for --.

* 1. Comma operator

Has the least precedence so it gets evaluated at the end. It evaluates the left operand expression and then the right one and returns the right one.

For ex.:

int x{2};

int y{3};

std::cout<<(++x,++y);

prints 4.

or

int z;

z= (a,b); //(a,b) gets evaluated and b is returned then z=b is evaluated.

z= a,b; // z=a is evaluated then b is returned and discarded;

It also acts as a seperator

constexpr int x{3}, y{2};

and for function call too.

* 1. Unary Operators

Arity 1. Such as -5.

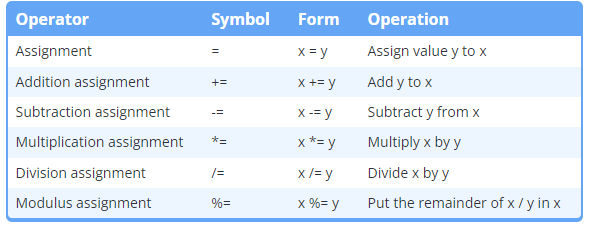
2 ops., Unary plus and minus. So +5 and -5.

* 1. Binary Operators

Arity 2. Such as 2 – 5.

5 ops., +,-,\*,/,%

All the binary operators have their equivalents in assignment ops.



* + 1. Binary Division

Division can be thought of as having 2 modes, integer and floating division. By default it does int div. but if a floating value is present it does floating division.

7/4 will return 1

7.0/4 will return 1.75

Simple way to return a FP Div. is

int x{2};

int y{3};

std::cout<<static\_cast<double>(x)/y;

prints 0.666…

We have convert value to FP if it’s going to be assigned to an FP type to trigger FP div.

For ex.:

int x{4};

int y{2};

double d{y/x};

stores 0 in d because y/x is in int div. and returns 2/4 = 0. Then this 0 is promoted to type double.

Instead, we have to convert either operand to double to trigger FP div., in which case the expr. evaluates to 0.5 and is directly assigned to d.

* 1. Ternary Operators

Arity 3. The conditional operator a==b ? x : y; is the only Ternary operator.

It is advised to parenthesize conditional expressions as they may have lower precedence than other operators such as <<.

Ternary ops. evaluate as expressions so they can be used to initialize variables and other places that expect expressions. They return a value. However this requires the type of the operands be either same or implicitly convertible.

For ex.:

constexpr int x{3};

std::cout<< (x>2) ? x : “yo”;

is an error as x and “yo” aren’t the same type and “yo” isn’t implicitly convertible to int.

* 1. Nullary Operators

Arity 0. throw is the only NO.

* 1. Logical Ops

The usual &&, || and !

Additionally C++ supports keyword versions of these ops.

They’re ‘and’, ‘or’ and ‘not’.

C++ supports short-circuit evaluation meaning if we have an expr.

if (false && true) {

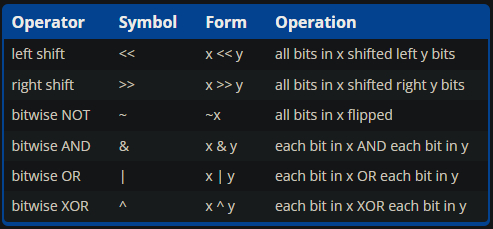
…

}

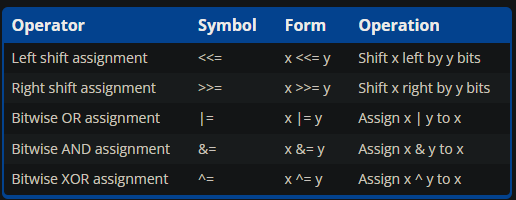
like so then it will eval the first expr. and immediately skip the rest as it knows AND needs both expr. to be true so it will be false regardless.

* 1. Bitwise Ops

Booleans or rather bits can be operated on by using these ops.



and they have assignment equivalents too.



* 1. Operator Overloading

We can define behavior of an operator manually as well in C++. This is because operators are implemented as functions internally, and we can use the concepts of functions on operators such as function overloading, called operator overloading for an operator, allowing us to define different behavior with the same function-name/operator. The function signature for an operator is

<return type> operator<operator>(<operands>) {…}

In any expression, the operands of operators decide where C++ will look for the operator’s overload. If both operands are fundamental types then it looks for a built-in overload, otherwise if one of them is a PDT then it looks in the PDT’s definition for an overload and if not found then it tries to numerically promote or convert the operand.

Ops we can’t overload: casting, sizeof, ‘?:’, ‘::’, ‘.’, ‘.\*’ and typeid.

Can’t create new ops either.

Any user-defined op. overload must have atleast 1 operand as a user-defined type.

Can’t change no. of operands for any operator.

Op. precedence can’t be changed.

There are 3 ways to overload an op.

Member Function Operator Overloading

Friend Function Operator Overloading

Normal Function Operator Overloading

* + 1. FFOO

We define the op. overload as a friend function.

For ex.:

class X{

private:

int x;

public:

X(int value): x{value}

{

}

friend X operator+(const X&, const X&);

};

X operator+(const X& a, const X& b) {

return a.x+b.x;

}

int main() {

X a{2};

X b{3};

X c{a+b}; //works

}

The order matters in operators, here a is the left operand so it is the 1st arg to the operator+ function, and b is the right operand so it is the 2nd arg.

This also means that 2 op. overloads, just like function overloads, are differentiated on the basis of their params and their order.

For ex.:

class X{

private:

int x;

public:

X(int value): x{value}

{

}

friend X operator+(int, const X&);

friend X operator+(const X&, int);

};

X operator+(int a, const X& b) {

return a+b.x;

}

X operator+(const X& a, int b){

return a.x+b;

}

int main() {

X a{2};

X b{a+2}; //calls the operator+(int,const X&)

X c{2+a}; //calls the second one

}

Here both functions have the same params but different order so they are different.

In FFOO, one operator overload can use the other one.

This is because a friend function can call another friend function.

Overloading Output stream Operator

For ex.:

#include<iostream>

class X{

private:

int x;

public:

X(int value): x{value}

{

}

friend std::ostream& operator<<(std::ostream&, const X&);

};

std::ostream& operator<<(std::ostream& out, const X& a) {

out<<a.x; //works, because the operator << is already overloaded for int.

return out; //to allow ‘<<’ chaining

}

int main() {

X a{2};

std::cout<<a<<’\n’; //works. Passes std::cout and a to operator<<.

}

There are 2 reasons for returning std::ostream&, first is that this allows us to chain the operator<< and second is that std::ostream specifically disallows copy-by-value so only the reference can be passed and returned.

Similarly we can overload operator>> with std::istream&.

Not every operator can be overloaded as friend functions,

‘=’, ‘[ ]’, ‘()’, ‘->’ can’t be FFOO but only MFOO.

* + 1. NFOO

For ex.:

class X{

private:

int x;

public:

X(int value): x{value}

{

}

int getX() {

return x;

}

};

X operator+(const X& a, const X& b) {

return a.getX()+b.getX();

}

int main() {

X a{2};

X b{3};

X c{a+b}; //works

}

This is the exact same thing as FFOO, the only difference is that as a normal function, it can’t see the private members of the object.

Whilst Normal Functions for Operator Overloading work just like FFOO, care must be taken to ensure the definition is visible to all the source files that use the operator’s overload.

That is, the function operator+(X&,X&) up there must be forward declared for all files that use operator + in X+X. To do so, we can forward declare the function in the file X.h and define it in X.cpp and that way we can use operator + in X+X anywhere X.h is #included.

* + 1. MFOO

Just like FFOO but the function is a member function, and that also means the leftmost operand hence param is automatically and implicitly the \*this and the rest params are the rest operands.

For ex.:

class X{

private:

int x;

public:

X(int value): x{value}

{

}

X operator+(const X& a)

{

return {this->x + a.x};

}

};

int main() {

X a{2};

X b{3};

X c{b+a}; //works

}

We can use the \*this or just use the members directly, either works. One thing to note is that MFOO, just like FFOO can see the members of other objects too, this is because access specifiers are applied across classes not objects as explained earlier in the doc.

Just like FFOO can’t be applied to all ops for a PDT, MFOO can’t be either. Such as for operator << and operator >>, because these operators require the left operand / first param to be std::ostream& or std::istream& and in the case of MFOO, the first param is the \*this. So these must be defned as NFOO or FFOO.

Similarly, whilst we can overload operators like ‘+’ to have left operand and first param to be the PDT itself, we can’t do so if the left operand or first param isn’t the PDT itself, for MFOO. Like X& operator+(int) as a MFOO requires left operand / first arg to be X& like X&+2 and won’t work for 2+X& in MFOO.

MFOO’s are recommended for ops like Unary. For a unary operator

class X{};

X operator+( ) {…}

is an example.

* + 1. Spaceship Operator <=>

Manually writing each comparison operator overload for a PDT is tedious, overloading this op allows us to simplify it.

* + 1. Prefix and Postfix increment/decrement

There are 2 versions of increment++/decrement-- operators.

The prefix version is --<identifier> and postfix is <identifier>--.

C++ differentiates the 2 using param list,

for prefix

#include<iostream>

class X{

public:

int x;

public

X(int v1):x{v1} {

}

X& operator--();

X operator--(); //notice the return type is X not X&

};

X& X::operator--( ) { //prefix decrement.

--x;

return \*this; //ok, simple enough

}

X X::operator--(int) { //unnamed/dummy parameter int is required. postfix decrement.

 X temp{\*this}; //make copy of X&, this converts the \*this to X& to X.

--(\*this); //prefix decrement on X

return temp; //return the copy instead

}

int main() {

X x{4};

std::cout<<(x--).x<<’\n’; //calls the postfix version, prints 4

std::cout<<x.x<<’n’; //prints 3

std::cout<<(--x).x; //prints 2

}

Prefix is simple as it just needs to modify the state and return the original object so it can return the reference. But postfix requires the state before the modification to be returned, and then also apply the modification, to do this a copy of the object must be made, then the state of the original object can be modified but not returned. This means x-- up there returns a new object entirely, and for this reason postfix is always going to be a bit slower than prefix in terms of performance. This is also true for fundamental types.

* + 1. Subscript Operator [ ]

For ex.:

class X{

private:

int x[10];

public:

X(): x{ }

{

}

int& operator[](int);

};

int& X::operator[](int index) {

return x[index];

}

int main() {

X a{};

a[0] =2; //ok

a[0]; //returns 2

}

[ ] operator returns a reference of the underlying type. This is because it is used in both assignment (as a left operand) and retrieval. It also has a higher precedence than assignment ‘=’ so it is evaluated first. By returning a reference, the compiler ensures that when the left operand of = is using operator[ ] then it will always return an l-value reference and is hence assignable.

But the same limitation is not true when the operator[ ] is using a const operand as const operands prohibit modification, so in that cast it can return by value as modification is not allowed anyway.

For const [ ], we use the const function keyword.

For ex.:

class X{

private:

int x[10];

public:

X(): x{ }

{

}

int& operator[](int);

int operator[](int) const; //const keyword differentiates in overload resolution

};

int& X::operator[](int index) {

return x[index];

}

int X::operator[](int index) const {

return x[index]; //implicitly converts the int& to int then returns

}

int main() {

X a{};

a[0] =2; //ok

a[0]; //returns 2

const X b{};

b[0]; //ok returns 0

b[0] =2 ;//error

}

[ ] is affected by pointer to object, that is if we have

X\* x{new X{}};

then calling x[0]; is an error as that requires [ ] to be overloaded for X\*. We can use

(\*x)[0]=2;

though.

The param can be any type.

* + 1. Parenthesis Operator ()

This operator is special because it allows us to take any number of operands.

For ex.:

class X{

private:

int x;

public:

X(): x{0}

{

}

void operator()(int);

int operator()();

};

int X::operator()(int a) {

return x+a;

}

void X::operator()(){

//do something

}

int main() {

X a{};

a(2); //ok, returns 2

a(3); //returns 3

a(); //does something

}

This op allows us to do something a normal member function would do, but without the function name.

Commonly used to implement functors, function objects, basically they share the state (as the state is in the class object) so we can see how our own class can implement functionality like lambda.

* + 1. Overloading typecasts

We can define our own typecasts too, that is, define how a given PDT would be casted into an object of another type.

The syntax is like so

operator <Type>() {<return value of Type type>}

For ex.:

class X{

private:

int x;

public:

X(): x{0}

{

}

operator int() const {return x;}

};

int main() {

X a{};

a+2; //works

}

a+2 works because a is implicitly casted to int as operator + here takes 2 ints and overload resolution finds that a can be converted to the other type, i.e. int, so it converts and then passes int, int to operator +. We can overload any typecast so even another PDT can be used in place of int as well.

The return type of this method isn’t defined, and C++ expects the correct type to be returned but doesn’t enforce it. This method doesn’t take any params either.

Overloading a typecast also allows typecasting using the casting operators, so we could use static\_cast<int>(x); in the example and that would work too.

* + 1. Copy Assignment Operator Overload

We do so for a PDT by overloading the assignment operator ‘=’.

This is different from the copy ctor, whilst both of them copy one object into another,

the copy ctor is invoked when a new object has to be created,

and the copy assignment operator is invoked when a new object doesn’t have to be created such as when an object already exists and we simply want to copy another object into it.

For ex.:

class X{

private:

int x;

public:

X(): x{0} {

}

X(const X& a): x{a.x} { //copy ctor

}

X& operator=(const X& a) {

if(this==a) //required

return \*this;

this.x= a.x;

return \*this;

}

};

int main() {

X a{};

X b{a}; //invokes copy ctor

X c{};

c=a; //invokes copy assignment operator.

}

The line this==a prevent self-assignment, that is, C++ allows

X x{};

x=x;

Whilst this does nothing in my example, it could cause issues if the underlying type is a pointer/array/etc. and we clear elements first before copying new ones, but if they are cleared and the object is the same then this means nothing gets copied/garbage value. So we avoid self-assignment in copy assignment.

This is not required in copy ctor because the ctor can only be invoked on new object and new object can’t self-assign without existing already.

It’s not necessary for the param to be const X&, it can be X& as well.

Just like default copy ctor, C++ also provides a default public copy assignment operator overload for our types, and just like copy ctor it does memberwise copy assignment. We can disable it with the delete keyword.

Shallow vs Deep Copy

Shallow copy is simple memberwise copy, but when our class uses dynamically allocated types in it then this implicit copy is not enough as it won’t copy the actual contents of the memory. Deep copy solves that but requires a lot more care, such as first deallocating the data in the object, then checking the other object has data and then allocating again and finally copying.

* + 1. Move Assignment

Works similarly to copy assignment operator overload but the major difference is that we modify the incoming object to not copy but move its contents into this object and in doing so transfer ownership of the data members to this object.

For ex.:

#include <iostream>

class X {

private:

int\* x;

public:

X() : x{new int{0}} {}

X& operator=(X&& obj) { // Move assignment, R-Value Reference

if (&obj == this) // if they are the same address

return \*this;

delete x;

x = obj.x;

obj.x = nullptr;

std::cout << "Move completed" << '\n';

return \*this;

}

X& operator=(const X& obj) { // Copy assignment

if (&obj == this) return \*this;

\*x = \*obj.x;

std::cout << "Copy completed" << '\n';

return \*this;

}

~X() {

delete x; // even if nullptr delete x, works

}

};

X yo() { return X{}; }

int main() {

X x1{};

X x2{};

x2 = yo(); //prints Move Completed

x2 = x1; //prints Copy Completed

}

x2=yo(), here x2 owns the members that were in yo(). x2 will then have address of yo()’s X’s member after x2’s own member is deallocated. We don’t delete obj.x in move assignment because if we did then even though we have copied the address already, the value at it would be deallocated. We point it to nullptr because if the other object is destroyed later, it will not deallocate the ‘moved’ pointer’s value.

Here yo() invokes the move assignment automatically because the object X{} will be deallocated after the end of the scope, so it is more efficient to move the members than copy them and delete the original. Moving is always more efficient than copying as the values theirselves remain where they are but the ownership(pointer) is the only thing transferred and this is true for all non-fundamental types but the downside is that at any given time, only 1 instance is active and the other instances don’t have ownership (pointer) over the data.

This is also an example of a simple class that creates a smart pointer as the pointer is wrapped in the class, RAII manages the class object’s lifetime and hence deallocates the pointer automatically.

The move assignment and move ctor are called when the functions are defined and the arg to assignment or construction is an R-Value. The only exception is when an automatic object is being returned by a function, in that case even if the returned value is an L-Value, it is moved.

In a specific case, the move assignment and move ctor are implicitly defined by the compiler.

This case is true only when, there is no user-defined copy ctor/copy assignment ops/move ctor/move assignment op/destructor.

1. Exception Handling

Done using 3 keywords,

throw: ‘raises’ or ‘throws’ an exception, i.e., a signal that something has gone wrong. In C++, throw can throw any type of value.

try: starts a block, this block is special as it can discover any of the exceptions raised by the code in the block.

catch: Also starts a block, which is always immediately after a try block and handles exceptions raised inside the try block. catch also requires a param list much like a function and handles exceptions that match that param list.

For ex.:

try {

throw 2;

}

catch (int a) {

//a has value 2 here

}

There can be multiple catch blocks, each handling an exception of the type its params accept.

The flow is like so, try block begins and runs normally until some exception occurs, if an exception occurs exits the try block immediately and go to the respective catch block, if there is none then it raises the exception and doesn’t continue code after the try catch blocks, if there is then goes to the catch block and executes code there then resumes code after the try catch blocks.

Try blocks catch exceptions not only from the code in the body but also if it leads to somewhere else in the code through function calls. This is due to stack unwinding.

For ex.:

void pp(){

throw 2;

}

int main() {

try {

pp();

}

catch(int) { //don’t need to use the exception object, here it is not copied at all

}

}

works

Stack Unwinding

When an exception occurs, the executor first looks if the code is in an immediate try block and there is a handler for the exception (a catch block for it), if not then it checks the same for the function’s call-site (it goes up a level in the call-stack), and so on until it finds a call-site under a try block with a matching handler or all the functions in the call-stack are checked and no handler is found. In the former case it jumps to that try block’s catch block after it ‘unwinds’ the stack, i.e., removes all functions (and destroys all variables etc. in it whilst doing so) that couldn’t handle the exception sequentially from the exception causing function till the function that does handle it, in the latter case it may or may not unwind the stack.

When an uncaught exception occurs, std::terminate() is called and stack may or may not be unwound (to preserve call-stack status to allow debugging and finding the source of the exception) and that means the variables and members may not be destroyed. In this case it is upto the OS to handle the program’s crash.

Try catch blocks can be nested.

The catch-all block

This can catch all types of exceptions,

try {

throw 2;

}

catch (…) {

}

The ellipsis are used for this.

The catch blocks are resolved sequentially from top to bottom and require the type of the exception to either exactly match the param or be a base class of the type.

This means,

#include <iostream>

int main() {

try {

throw 'a';

}

catch (double) {

std::cout<<"double"<<'\n';

}

catch(int) {

std::cout<<"int"<<'\n';

}

}

leads to an uncaught exception as even though ‘a’ is assignable to double and int, neither types exactly match it (char) and neither are a base class of char.

It is recommended to catch exceptions of non-fundamental types by const reference to avoid copying the exception.

That is,

#include <iostream>

class X {

public:

X() = default;

X(const X& x) { std::cout << "X{} copy ctor" << '\n'; }

};

int main() {

try {

throw X{};

} catch (X) {

std::cout << "X()" << '\n';

}

}

prints “X{} copy ctor” then “X()” as X’s object is copied to the catch block.

* 1. Exception Handling with Classes

When a class/struct has an exception in the ctor itself, it immediately raises the exception just like normal functions. However, the dtor is not called.

For ex.:

#include<iostream>

class Y{

Y() { std::cout<<”Y()”<<’\n’; }

~Y() { std::cout<<”~Y()”<<’\n’;}

};

class X{

int a;

Y y;

X(): a{0}, y{} {

std::cout<<”X()”<<’\n’;

throw 2;

}

~X() {

std::cout<<”~X()”<<’\n’;

}

};

int main() {

try{

X x{};

}

catch(…) {}

}

prints “X()” then “Y()” then “~Y()” but it doesn’t call ~X() at all.

This is to say, the class’ dtor is never called but the data members are destructed nevertheless. This is problematic if the data members are dynamically allocated as they don’t get deallocated so care must be taken.

It is recommended to create a custom class that has members to contain details for exceptions and use that class for exception handling everywhere. This class can be extended too. In the case of an inherited exception class, when handlers are being resolved they will resolve to the base class if it appears before the class.

For ex.:

#include <iostream>

class X{};

class Y: public X {};

int main() {

try {

throw Y{};

}

catch(X) {

std::cout<<"X()"<<'\n';

}

catch(Y) {

std::cout<<"Y()"<<'\n';

}

}

prints X() even though the exception is of type Y. This is only true for PDTs. Here, it’d be better to have a const ref instead of raw class to avoid needless copy.

This is why it is recommended to have the most-derived class earlier in the catch-list.

* 1. std::exception

Defined in the #include<exception> this base class is used by all exception classes used in the Standard Library, so we can catch it to handle any exception being raised from the std.

We can throw std::exception too, or any of its derivative classes

<https://en.cppreference.com/w/cpp/error/exception>

std::runtime\_error(“string message”) is a derived class commonly used for custom exceptions.

We can extend std::exception too.

* 1. Rethrow

We can throw exceptions from a catch block too, in which case it makes the executor exit the current try catch block and look for handlers above it. This is called throwing, rethrowing is if we throw the same exception we caught, care must be taken with rethrowing it may be not the right type when classes are concerned.

For ex.:

int main() {

try {

try {

throw 2;

}

catch(int)

{

throw ‘a’; //throwing another exception, rethrowing would be throw <int>; here.

}

catch(char) //never ran

{

}

}

catch (char)

{

}

}

The catch(char) of outside block is triggered here.

* 1. Function Try Blocks

Try catch blocks can be made at the function level too.

For ex.:

#include <iostream>

void yo() try

{ //whole function block is a try block

throw 2;

}

catch(...) { //immediately after function block the catch blocks are started

std::cout<<"Function level try block"<<'\n';

}

int main() {

try {

yo();

} catch(...) {

std::cout<<"Normal try block"<<'\n';

}

}

prints “Function level…” because of the existence of a function-level try block.

The same works even for methods, the syntax is a bit different for class ctors. That is,

class X{};

class Y:public X{

public:

int a;

Y(int v1) try : a{v1}, X{} {

}

catch(...) {

}

};

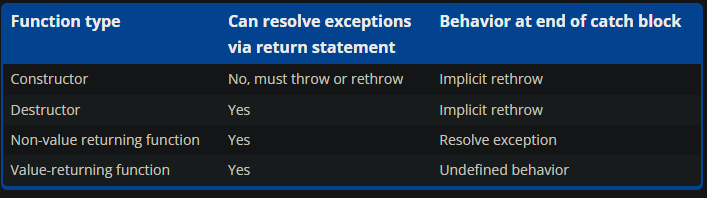
The try keyword appears before the member list initializer and base ctor call, that is to indicate if there is an exception in member initialization or in the base ctor then the catch block will be triggered just the same as if there is any exception in the ctor body.

Normal catch block vs. Function-Level catch block

Normal catch blocks can do whatever, return, rethrow, throw or do nothing with the control flow (in which case execution continues after the try catch blocks).

Function-level catch blocks must either rethrow or throw new exceptions but for ctors they can never return and when they do nothing with the control flow they implicitly rethrow.

This table shows the differences between different types of function-level catch blocks



Since this behavior is not the same everywhere, it is recommended to explicitly throw, rethrow or return from all function-level catch blocks to clearly indicate the intentions.

For ctors, we can’t call the dtors in the function-level catch block because the object isn’t even initialized so we need explicitly handle the members in the catch block.

* 1. Destructors should never throw

If they do then the executor immediately terminates the program.

* 1. Exceptions have a performance cost

Whenever exception handling is added it increases the binary size and at runtime causes a small performance overhead with stack unwinding and the additional checks and setups that must be made with the handlers.

There is a concept called Zero-Cost Exceptions in modern compilers which remove this overhead however incur an even larger penalty if there is an exception.

This is why exception handling must only be added when the issue occurs infrequently and the exception is not creating a code path for majority of runtimes (if a program uses an exception most of the time to do things then it is incredibly inefficient and prone to breaking) and there isn’t a simpler check for handling issues.

* 1. noexcept

Exception Specification is a language mechanism that helps determine what type of exception a function may throw. This is useful in some contexts such as in dtors where we want a way to know if a function is ‘safe’ to be called.

The noexcept specifier is one such spec,

Adding this specifier to a function’s declaration tells the callers that it is guaranteed this function will not throw, and on the contrary not having it means it may throw.

For ex.:

void yo() noexcept {}

int main() {

yo(); //use yo() as normal

}

noexcept doesn’t prevent the function from creating an exception, it expects there to be no uncaught exceptions going outside the function and if they do, std::terminate is called even if the function’s call-site is in a try catch block, furthermore stack unwinding may or may not occur in this case.

Just like return type of a function doesn’t take part in overload resolution, neither does noexcept. Meaning we can’t overload a function with the only change being noexcept status.

noexcept also accepts a Boolean parameter where noexcept(true) means the function is noexcept and noexcept(false) means it may throw.

Some functions are implicitly noexcept,

dtors, default/copy/move ctors, assignment ops copy/move, comparison ops (from C++20).

If the data member of a class has a ctor that is non-noexcept then the default/copy/move ctors are also non-noexcept. Similarly if any of these call a function which is non-noexcept either implicitly or explicitly then that function will be non-noexcept too.

On the contrary, user-defined ctors, normal functions and custom ops are non-noexcept by default.

noexcept as an operator

It can be used like an operator with 1 operand and returns the noexcept status of the operand. This is done at compile-time itself.

For ex.

void yo() noexcept(true) {}

int main() {

constexpr bool a {noexcept(1+2)}; //true, ints are noexcept

constexpr bool b {noexcept(yo())}; //true

}

noexcept also allows the compiler to perform some more optimizations and it may also keep the stack in an unwindable state for faster performance since the function guarantees to not throw. But it shouldn’t be used everywhere as it creates a no-throw guarantee (more guarantees here <https://www.learncpp.com/cpp-tutorial/exception-specifications-and-noexcept/>) that the code will never throw and going back on such a guarantee later would be anti-pattern.

Dynamic Exception Specification

This is a deprecated feature that was used before noexcept and indicates which exceptions a may directly or indirectly throw.

For ex.:

int doSomething() throw(); // does not throw exceptions

int doSomething() throw(std::out\_of\_range, int\*); // may throw either std::out\_of\_range or a pointer to an integer

int doSomething() throw(...); // may throw anything

1. Smart Pointer

We know traditional pointers have a myriad of ways in which they may lead to memory leaks, such as early returns, exceptions etc. which may fail to call the deallocator. This is where we can wrap our pointer in a class which will deallocate it because of RAII which automatically destroys objects and calls dtors at the end of the scope.

std::auto\_ptr

Deprecated in C++17 because of numerous issues it brought. It was a smart pointer but it implemented move semantics by default, meaning even its copy ctor and copy assignment also moved the pointer and that meant if the object was passed to functions etc. it would get moved and then die with the function. It also had non-array delete so whilst it allowed any type, it wouldn’t deallocate the dynamically allocated array. And because of these issues, it couldn’t be used in the standard library.

* 1. std::unique\_ptr

Defined in #include<memory>, this pointer wrapper only allows 1 owner of the pointer at any given time, this means it never copies, only moves its contents.

For ex.:

#include <iostream>

#include<utility> //for std::move

#include<memory>

int main() {

std::unique\_ptr<int> u1{new int{0}};

std::unique\_ptr<int> u2{};

u2=u1; //error

u2=std::move(u1); //ok

}

u2=u1 fails because it invokes copy assignment which is a deleted function in std::unique\_ptr and we have to invoke move to remove that error.

unique\_ptr overrides operator \* and operator -> so we can access the underlying object’s data like a normal pointer, furthermore it also typecast overload to bool which returns false if the underlying pointer is nullptr.

unique\_ptr is also able to understand and accept array types. However, std::array and std::vector should be preferred since they are meant to do this.

std::make\_unique

Introduced in C++14, this is a template function that creates a unique\_ptr and returns it using its type and args.

For ex.:

#include <iostream>

#include<utility> //for std::move

#include<memory>

int main() {

auto u1{std::make\_unique<int>(5)};

auto u2{std::make\_unique<int>()};

}

W3e can either directly give a value as an arg or leave it null for the unique\_ptr to have nullptr.

This method is recommended over manually defining std::unique\_ptr because of brevity and exception safety issue’s solution.

Exception Safety Issue

Consider a function call

myFunction(std::unique\_ptr<T>(new T{}), mySecondFunction());

In this call, the order of execution is not guaranteed and it may be that the new T{} is constructed first and then mySecondFunction() is ran, in which case if it throws and std::unique\_ptr still isn’t constructed (but its value is allocated and inside an anonymous object) then on stack unwinding the anonymous object won’t see a destructor call as it isn’t even constructed and that leads to memory leak. make\_unique doesn’t face this issue because the value inside it is constructed all at once.

Passing and Returning

Since unique\_ptr is a class by itself, it can be passed (by moving) to functions and returned from the functions too.

We can also use the underlying raw pointer and deal with it directly too, we get it using the .get() method of it.

For ex.:

#include <iostream>

#include<utility> //for std::move

#include<memory>

int main() {

auto u1{std::make\_unique<int>(5)};

auto u2{std::make\_unique<int>()};

}

prints 2.

Whilst unique\_ptr itself cannot be copied and hence lead to 2 unique\_ptr managing the same resource, the resource could be an L-Value and hence have its own lifetime so it can still be deallocated before the unique\_ptr ends. This is why care must be taken to use unique\_ptr correctly.

* 1. std::shared\_ptr

This smart pointer allows sharing specifically, i.e., whilst unique\_ptr

only allowed a single unique\_ptr to hold ownership, shared\_ptr allows multiple shared\_ptrs to do so. So copying is allowed here.

For ex.:

#include<memory>

int main()

{

std::shared\_ptr<int> s1{new int{2}};

{ //sub-block

auto s2{s1}; //copy an existing shared\_ptr

} //s2’s lifetime ends

std::cout<<\*s1<<’\n’; //works

auto s3{ std::make\_shared<int>(3)}; //also works

}

Here, even though s2’s lifetime ends and it should deallocate the memory, it doesn’t because it has a data member (a pointer, also called a control block) that tracks how many copies of the object are active at a time and cleans up the underlying pointer only if that is at 1. But at s2, it incremented it for s2 to 2 and when s2 was to be destroyed it checked and found it to be >1 and hence didn’t deallocate the underlying memory. Since this control block is a pointer itself, all copies of a shared\_ptr are updated at the same time and know exactly how many copies are active.

std::make\_shared is more performant than normal definition of shared\_ptr.

shared\_ptr can accept a unique\_ptr in its ctor, however the unique\_ptr must be moved. The other way is not safe but can be explicitly done too.

Up until C++20, shared\_ptrs didn’t have a proper support for arrays and mustn’t be used with C-Style arrays but from C++20 it is safe and allowed.

shared\_ptr, and normal pointers, suffer from an issue called circular reference. This is when a shared\_ptr points to another shared\_ptr, which points back to this shared ptr, then neither of them get deallocated because they need the next one to be deallocated first, this creates a cycle and none of them are deallocated by RAII.

For ex.:

#include <iostream>

#include <memory>

class B;

class A {

public:

std::shared\_ptr<B> b\_ptr;

A() {

std::cout << "A()" << std::endl;

}

~A() {

std::cout << "~A()" << std::endl;

}

};

class B {

public:

std::shared\_ptr<A> a\_ptr;

B() {

std::cout << "B()" << std::endl;

}

~B() {

std::cout << "~B()" << std::endl;

}

};

int main() {

auto a\_ptr{ std::make\_shared<A>() };

auto b\_ptr{ std::make\_shared<B>() };

a\_ptr->b\_ptr = b\_ptr;

b\_ptr->a\_ptr = a\_ptr;

}

Here ~B and ~A are never called because the shared\_ptr a\_ptr wants its count to drop to 1, and b\_ptr wants the same, however their internal states (data members) cross-reference each other and as long as they keep the count up, the outside shared\_ptrs can’t deallocate.

* 1. std::weak\_ptr

This smart pointer is meant to be used with shared\_ptr and solves the circular reference problem with shared\_ptr. It simply has a shared copy of the pointer of a shared\_ptr however it doesn’t increase the internal count of shared\_ptr and hence doesn’t prevent the shared\_ptr from deallocating.

For ex.:

#include <iostream>

#include <memory>

int main() {

auto s1{std::make\_shared<int>(3)};

std::weak\_ptr<int> w1{s1};

auto s2{w1.lock()};

std::cout<<\*s2<<'\n';

}

However, weak pointer can’t be used to access the underlying value, for that we need to use its .lock() method which returns a shared\_ptr, and hence increments the internal counter of the shared\_ptr the weak\_ptr references. However, unlike a normal shared\_ptr, since this shared\_ptr is created in the scope it is cleared with the scope and hence still doesn’t lead to a circular reference as that shared\_ptr is a data member and exists outside the scope too.

weak\_ptr and dangling reference

Since this smart pointer doesn’t increment the counter it is certain there will be situations where its underlying shared\_ptr has already been deallocated, it has a method .expired() which returns true if that is true.

1. Standard Library
   1. Containers

These are container classes that are present in the std.

There’s the sequence containers:

std::vector, std::deque (double ended queue), std::array, std::list (double linked list), std::forward\_list and std::basic\_string.

These containers hold a sequence of data.

Then there’s associative containers:

set, multiset, map, multimap.

These are containers that sort their data.

Container Adapters

There’s stack, queue and priority queue.

These containers ‘adapt’ to the uses, they use sequence containers but with some extra functionality, we can define the sequence container being used internally.

Stack is a container that allows LIFO and by default uses deque. Can also use vector or list or deque.

Queue is FIFO and and uses deque. Can use list or deque.

Priority Queue sorts the input and keeps the smallest element at the front, this element is also called its ‘highest priority’ element.

* 1. Iterators

Iterator is like a pointer that traverses over a container class but it overrides some ops and defines some methods.

\*op Returns the element

op++ Moves the iterator to the next element, op-- moves it back. Same for ++op and --op.

op== and op!= Simply checks if 2 iterators are pointing to the same element.

op= Assigns the iterator a new position in the container.

Member functions

begin() Returns an iterator at the first element of the container

end() same but for last + 1 so end() points to no value.

cbegin() Read only iterator like begin()

cend() Read only iterator like end()

All containers provide 2 iterators,

<container>::iterator and <container>::const\_iterator (read only iterator)

1. Debugging

Many ways to debug C++ files, we can use gdb to do so as well.

First we compile a file with debug symbols included (optional but important)

g++ -g -I ../Helpers/Easybench -o points PIP.cpp ../Helpers/Easybench/Easybench.cpp

Then we can debug using gdb with

gdb points

This starts the gdb debugger on the points file and presents a shell.

In the gdb we can,

catch throw

stops the debugger on any exception at runtime.

run

runs the file

break PIP.cpp:25

In the points compiled file, find the PIP.cpp source file and set a breakpoint at line 25. Then if we run and line 25 is executed it pauses the debugger at that line and we can do next or step or continue as in normal debuggers.

We can also use

backtrace

to print the stacktrace until this point, or

print my\_var

to print the value of a variable. And we can also do

watch my\_var

which sets a watchpoint on a variable and pauses debugging whenever the value of my\_var changes.

We can also do

break PIP.cpp:25 if my\_var>10

sets a conditional breakpoint.

And lastly,

help

or

help <somecommand>

to get the help about all or a specific gdb command.

For ex.:

