Beginning C++ Programming - From Beginner to Beyond

David Corzo

2020 May 18

Contents

Chapter 1

Introduction

1.1 Why learn C++?

- Popular:
  - Lots of code is still written in C++.
  - Programming language popularity indexes ranks C++ high.
  - Active community, GitHub, Stack overflow.
- Relevant:
  - Windows, Linux, MaxOSX, Photoshop, Illustrator, MySQL, MongoDB.
  - Amazon, Apple, Microsoft, PayPal, Google, Facebook, MySQL, Oracle, HP, IBM, more...
  - VR, Unreal Engine, Machine learning, networking & telecom, more...
- Powerful:
  - Super-fast, scalable, portable.
  - Supports both procedural and object-oriented programming.
- Good career opportunities:
  - C++ skills always in demand.
  - C++ = Salary++.

1.2 Modern C++ and the C++ standard

- Early 1970s: C programming language; Dennis Ritchie.
- 1979: Bjarne Stroustrup; 'C with classes'.
- 1983: Name changed to C++.
- 1989:First commercial release.
- 1998: C++98 Standard.
- 2003: C++03 Standard.
- 2011: C++11 Standard.
- 2014: C++14 Standard.
- 2017: C++17 Standard.

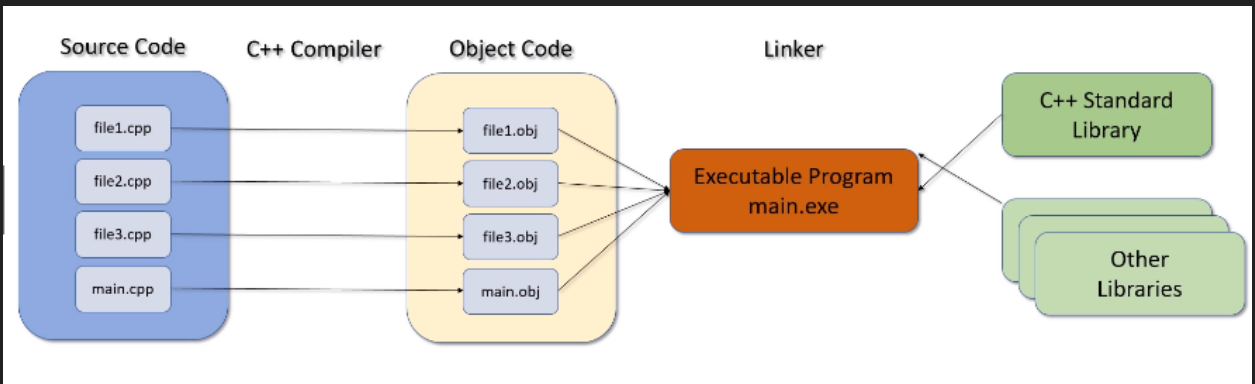
1.2.1 Modern C++ and C++ Standard

- Classical C++: Pre C++11 Standard.
- Modern C++:
  - C++11: Lots of new features.
  - C++14: Smaller changes.
  - C++17: Simplification.

1.3 How does it all work?

- Use non-ambiguous instructions.
- Programming language: source code, high level, for humans.
- Editor: text editor. *.cpp* and *.h* files.
- Binary or other low level representation: object code for computers.
- Compiler: Translates from high-level to low-level.
- Linker: links together our code with other libraries, creates *.exe*.
- Testing and debugging: finding and fixing program errors.

1.3.1 The C++ build process



1.3.2 Integrated Development Environments (IDEs)

- Editor.
- Compiler.
- Linker.
- Debugger.
- Keep everything in sync.

IDEs

- CodeLite.
- Code::Blocks.
- NetBeans.
- Eclipse.
- CLion.
- Dev-C++.
- KDevelop.
- Visual Studio.
- Xcode.

Chapter 2

Installation and setup

2.1 Installing C++ Compiler for windows

- Go to: `http://mingw-w64.org/doku.php/download/mingw-builds`
- Go to: Downloads, find the build, download and run executable.
- Set the environment variable:
  - Control panel → Edit system environment variables.
  - Environment variables → System → Path → Edit.
  - New → Browse → < go to your instaltion dir > → OK.
- Go to CMD: type `c++ --version` → Should print version.

2.2 VSCode Project Setting Up

Inside the .vscode directory add:

- `c_cpp_properties.json`:

```
{
    "configurations": [
        {
            "name": "Win32",
            "includePath": [
                "${workspaceFolder}/**"
            ],
            "defines": [
                "_DEBUG",
                "UNICODE",
                "_UNICODE"
            ],
            "browse": {
                "path": [
                    "${workspaceRoot}",
                    "C:\\Program Files\\mingw-w64\\mingw64\\bin\\gcc.exe"
                ]
            },
            "compilerPath": "C:\\Program Files\\mingw-w64\\mingw64\\bin\\gcc.exe",
            "cStandard": "gnu18",
```

```
            "cppStandard": "gnu++14"
        }
    ],
    "version": 4
}
```

- launch.json:

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "g++.exe - Compilar y depurar el archivo activo",
            "type": "cppdbg",
            "request": "launch",
            "program": "${fileDirname}\\${fileBasenameNoExtension}.exe",
            "args": [],
            "stopAtEntry": false,
            "cwd": "${workspaceFolder}",
            "environment": [],
            "externalConsole": false,
            "MIMode": "gdb",
            "miDebuggerPath": "C:\\Program Files\\mingw-w64\\mingw64\\bin\\gdb.exe",
            "setupCommands": [
                {
                    "description": "Habilitar la impresión con sangría para gdb",
                    "text": "-enable-pretty-printing",
                    "ignoreFailures": true
                }
            ],
            "preLaunchTask": "C/C++: g++.exe build active file"
        }
    ]
}
```

- In order to establish the formatting style put the following in `settings.json`:

```
{
    "C_Cpp.clang_format_fallbackStyle": "{ BasedOnStyle: LLVM, UseTab: Never, IndentWidth: 4, TabWi
    "emmet.showSuggestionsAsSnippets": true,
    "files.associations": {
        "*.rmd": "markdown",
        "iostream": "cpp"
    }
}
```

- task.json:

```
{
    "version": "2.0.0",
    "tasks": [
        {
            "label": "C/C++: g++.exe build active file",
            "type": "shell",
            "command": "C:\\Program Files\\mingw-w64\\mingw64\\bin\\g++.exe",
            "args": [
```

```
                "-g", "main.cpp", "-o", "a.exe" // , "&&", "main"
            ],
            "problemMatcher": [
                "$gcc"
            ],
            "presentation": {
                "echo": false,
                "reveal": "always",
                "focus": false,
                "panel": "shared",
                "showReuseMessage": true,
                "clear": false
            },
            "group": {
                "kind": "build",
                "isDefault": true
            }
        }
    ]
}
```

Chapter 3

Curriculum Overview

3.1 Curriculum overview

Get started quickly programming in C++.

- Getting started.
- Structure of a C++ program.
- Variables and constants.
- Arrays and vectors.
- Strings in C++.
- Expressions, Statements and Operators.
- Statements and operators.
- Determining control flow.
- Functions.
- Pointers and references.
- OPP: Classes and objects.
- Operator overloading.
- Inheritance.
- Polymorphism.
- Smart pointers.
- The Standard Template Library (STL).
- I/O Stream.
- Exception Handling.

Chapter 4

Getting Started

4.1 Writting our first program

- Create a project.
- Create a file and type the following code.
- This program is going to take in a number and then display "Wow that is my favorite number".

```
#include <iostream>
int main() {
    int favorite_number; // stores what the user will enter.
    std::cout << "Enter your favorite number between 1 and 100"; // prints.
    std::cin >> favorite_number;
    std::cout << "Amazing!! That's my favorite number too!" << std::endl; // prints that line. endl adds
}
```

4.2 Building our first program

- Building involves compiling and linking.
- In vscode you can run the compile task by pressing ctrl+b.
- Linking means grabbing all the dependencies the main function needs, making .o or object files and adding them to the executable or the .exe.
- Tipically modern compilers have the option to not produce the object files and go ahead and just produce a single executable. IDEs tipically also hide the object files if they are produced.
- By *cleaning* a project what we mean is the object files are deleted and an executable will be produced.

4.3 What are compiler errors?

- Programming languages have rules.
- Syntax errors: something wrong with the structure.

  ```
  std::cout << "Errors << std::endl; // the string is never terminated.
  ```

- Semantic errors: something wrong with the meaning:

  ```
  a + b; // to sum a and b when it doesn't make sense to add them, maybe they are not numbers for exa
  ```

4.3.1 Examples of errors

Not enclosing a string with the ” characters.

```
int main() {
    std::cout << "Hello world << std::endl; // string is not terminated with the other ".
    return 0;
}
```

Typos in your program:

```
int main() {
    std::cout << "Hello world" << std::endll; // endll doesn't exist, this is syntaxis errors.
    return 0;
}
```

Missing semi-colons:

```
int main() {
    std::cout << "Hello world" << std::endl // missing semicolon.
    return 0;
}
```

Function doesn’t return the type specified, in this case the function doesn’t return an integer.

```
int main() {
    std::cout << "Hello" << std::endl;
    return; // main needs to return an integer and it is returning a void.
}
```

Not returning the specified type.

```
int main() {
    std::cout << "Hello World" << std::endl;
    return "Hello"; // "Hello" is not an integer. Error.
}
```

Missing Curly brace:

```
int main() // opening curly brace missing.
    std::cout << "Hello World" << std::endl;
}
```

Semantic error (example: adding something when it doesn’t make sense).

```
int main() {
    std::cout << ("Hello world" / 125) << std::endl; // dividing a string by a number, this doesn't mak
    return 0;
}
```

4.4 What are compiler warnings?

- It is good practice to never ignore compiler warnings.
- The compiler will recognize a potential issue but is still able to produce object code from the source code, things such as uninitialized variables.
- It’s only a warning because the compiler is still able to generate correct machine code.
- Example:

```cpp
int miles_driven; // never initialized, this value could be anything.
std::cout << miles_driven << std::endl;
/* Warning: 'miles_driven' is used uninitialized in this function. */
```

- Another example is when you declare variables and never use them.

```cpp
int miles_driven = 100;
std::cout << "Hello world" << std::endl;
/* Warning: unused variable 'miles_driven'. */
```

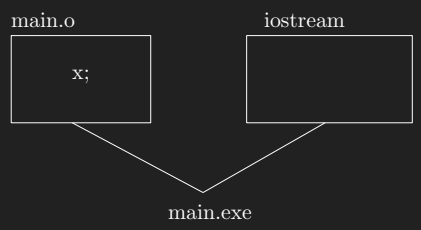
- As a rule you want to produce warning free source code.

4.5 Linked errors

- The linker is having trouble linking all the object files together to create an executable.
- Usually there is a library or object file that is missing.

4.5.1 Example

```cpp
#include <iostream>
extern int x; // this means the variable is defined outside this file.
int main() {
    std::cout << "Hello world" << std::endl;
    std::cout << x;
    return 0;
}
/* This program will compile, but in runtime you will get a linker error. */
```



According to the linker x is undefined, thus an error is produced.

4.6 Runtime Errors

- Errors that occur when the program is executing.
- Some typical runtime errors include:
  - Divide by zero.
  - File not found.
  - Out of memory.
- Can cause your program to crash.
- Exception handling can help deal with runtime errors.

4.7 What are logic errors?

- Errors or bugs in your code that cause your program to run incorrectly.
- Logic errors are mistakes made by the programmer.

Suppose we have a program that determines if a person can vote in an election and you must be 18 years or older to vote.

```
if (age > 18) { // This means that age cannot be 18 thus 18 yearolds would not be able to vote. 18 is n
    std::cout << "Yes you can vote" << endl;
}
```

4.8 Section challenge solution

```
#include <iostream>
int main() {
    int favorite_number;
    std::cout << "Enter your favorite number between 1 and 100: ";
    std::cin >> favorite_number;
    std::cout << "Amazing!! Thats my favorite number too!" << std::endl;
    std::cout << "No really!!, " << favorite_number << " is my favorite number!" << std::endl;
}
/* Output:
Enter your favorite number between 1 and 100: 67
Amazing!! Thats my favorite number too!
No really!!, 67 is my favorite number!
*/
```

Chapter 5

Structure of a C++ program

5.1 Overview of structure of a C++ program

- C++ has lots of keywords.
- Compared to other languages:
  - Java has about 50.
  - C has 32.
  - Python has 33.
  - C++ has 90.
- You can view these words in the following link: c++ reference.
- Keywords cannot be redefined nor used in a way they are not intentionally made for. This is why variables have naming conventions for example.
- Difference between keywords and identifiers, a keyword is something that is built in the programming language, a keyword has specific purpose and meaning that cannot be changed; identifiers are defined and user for the programmers purposes, a variable name is an example of an identifier, another example are function names, there are rules for identifiers, also conventions.
- C++ also has punctiation, and you must adhere to the punctuation rules, things such as "", <<, >>, ;, etcetera, are examples of punctuation.
- When you assemble rules of punctuation, keywords, identifiers, rules, you end up with *syntax* this refers to rules and conventions that you must follow.

5.2 #include Preprocessor directive

- What is a preprocessor?
  - A preprocessor is a program that processes your source code before your compiler sees it. C++ preprocessor strips all the comments from the source code and then sends it to the compiler, it replaces each comment with a single space.
- Directives in C++ start with a # sign. Examples include:

```
#include <iostream>
#include "myfile.h"
#if
#elif
```

```
#else
#endif
#ifdef
#ifndef
#define
#undef
#line
#error
#pragma
```

  - In simple terms the compiler replaces the include directive with the file that its refering to, then it recursively processes that file as well.
  - By the time the compiler sees the source code it has been stripped of all comments and all preprocessor directives have been processed and removed.
  - Preprocessor directives are routinely used to conditionally compile code, for example say I want to execute some piece of code only if the program is running in the windows operating system and execute some other if the program is being run by MacOS, preprocessor directives can check to see if you are on windows and run your code, if you're not on windows then the preprocesor directive will run the MacOS code.
  - The C++ preprocessor does not understand C++. It simply processes the preprocessor directives and gets the code ready for the compiler, the compiler is the program that does understand C++.

5.3 Comments in C++

- Comments are programmer readable explanations in the source code; explanations, notes, annotations, or anything that adds meaning to what the program is doing.
- The comments never makes it to the compiler, the preprocessor strips them, the comments are just to enhance the readibility of your source code, it's intended for humans and the compiler never sees it.
- There are basically two ways of writing comments, single line comments and multiline comments.
- Single line comments start with `//`, the remaining characters in that line are considered a comment by the preprocessor.

  ```
  int var = 100; // This is a comment.
  ```

- Multiline comments start with `/*` anything after these characters up until `*/` are considered a comment by the preprocessor. Everything between the `/* */` will be ignored.

  ```
  /* This is a line.
  This is another commented line.
  This is yet another.
  ...
  */ int var = 100;
  ```

- The idea behind comments are to make self documenting code. Self documenting code is the practice of wrinting down what your code does and how it does it so that other people may understand it.

  ```
  int var = 100; // I'm initializing a variable to 100 in order to use <where I will use it> for <sta
  ```

  - Keep in mind not every single line needs commenting, comments must be used only when they are needed, sometimes code can be very hard to read and that's were you want to use comments. You don't want to write a comment for every line of code saying what you do because what you are doing is easily and clearly dedusable, other code however can become very unreadable and complicated and that's were you might want to use comments so that other people know what you are doing. Don't comment the obvious.

- As a rule: don't comment the obvious, good commenting doesn't justify bad code, and if you made changes to code make sure you also change the comments to save time and confusion, keep comments in sync.

5.4 The main() function

- Every C++ program must have exactly one `main()` function.
- main() is the starting point of program execution.
- the `return 0;` statement in the `main()` function indicates successful program execution.
- When a C++ program executes the operating system calls the `main()` and the code between the curly braces executes. When execution hits the return statement, the program returns the integer 0 to the operating system, if the return value is 0 then the program terminated sucesfully, if the value returned is not 0 then the operating system can check the value returned and determine what when wrong.
- There are two posible ways of writing the main function. They work almost the same and the differences are suttle. All the program actions are contained in the main function.

```
// First way:
int main(/* No parameters. */) {
    // <code>
    return 0;
}
// Second way:
int main(int argc, char *argv[]){
    // <code>
    return 0;
}
```

  - The first version expects no information to be passed in from the operating system in order to run. This is the most common version.
  - The second version allows for parameters to be passed in to the main function when its called, so for example you can pass in an integer or a string that you can then use in your program when it's called from the command line. You would call your program like this in the command line: `program.exe n arg1 arg2 argn` the first argument or `int argc` must be the number of arguments that you will pass in, the next arguments are delimited by spaces and collected and stored inside your program in the `char *argv[]` array which you can use in your program respectively.
  - It's important to see the distinction in order not to get confused when you are looking at code out on the internet.

5.5 Namespaces

- As C++ programs become more complex the combination of our own code, the C++ standard library, and other third party libraries, sooner or later C++ encounters two names repeated, and at that point C++ doesn't know which one to use. This is called a naming conflict and its described when company X named something the same as company Y.
- This is where namespaces come in, you can specify which library you are refering to by using the name of the library and the scope resolution operatior (`::`)
- C++ allows developers to use namespaces as container to group their code entities in to a namespace scope.

- An example is the `std::cout` statement, tecnically it is saying to the C++ compiler to search in "std" or the standard library the function "cout" and use that one.
- Namespaces are introduced to reduce the posibility of a naming conflict.
- However, it can get tedious to write the library name, then the scope resolution operator and finally the function name; thus you can use the `using namespace <insert library name>;` statement to specify that any function from there on will come from the library spefied, now C++ knows which namespace you are using.

```
#include <iostream>
using namespace std;
int main() {
    // <code>.
    return 0;
}
```

  - C++ in the code above knows the moment we said `using namespace std;` that any function referenced from that point will be refering to the std library function.

- However there is still a problem, `using namespace std;` doesn't just state to use cout and cin, it brings a lot of other functions we might not know about, for this we can explicitly say we just want to use a certain function of a certain library.

```
#include <iostream>
using std::cout;
using std::cin;
int main() {
    // <code>.
    return 0;
}
```

  - You can still use `cout` and `cin` without having to write `std::cout` and `std::cin` and we are not getting any other names from the standard library of which we won't need. In larger programs its best practice to declare the functions you will use manually and not the namespace.

5.6 Basic input and output

- `cout, cin, cerr, clog` are objects representing streams. They are defined in the iostream library.
- `cout`:
  - It is a standard output stream.
  - Defaults to the console.
- `cin`:
  - It is a standard input stream.
  - Defaults to the keyboard.
- `cerr`:
  - It is a standard error stream.
  - Defaults to the console with the error stream.
- `clog`:

  - It is a standard log stream.
  - Defaults to the console with the log stream.

- `<<` is the insertion operator:
  - Used on output streams.
- `>>` is the extraction operator:
  - Used on input streams.

5.6.1 `<<` with `cout`

- The insertion operator (`<<`) inserts data into the cout stream. It inserts the value of the operad(s) to its right, this operand can chain various variables or objects.

```
cout << data;
```

- Chaining multiple insertion in the same statement:

```
cout << "data 1 is " << data1 << endl;
cout << "data 1 is " << data1 << "\n";
```

- It is important to understand that the insertion operator does not automatically add line breaks to what is printed, that can be added by `endl` or by explicitly adding it using the newline escape character `"\n"`. The `endl` manipulator it will also flush the stream, this is important because if the stream is buffered the program might not print anything until its flushed.

5.6.2 `>>` with `cin`

- Used to extract data from the cin (which defaults to the keyboard) stream based on the data's type and then stores the information into the variable to the right of the extraction operator.

```
cin >> data;
```

- It can be chained.

```
cin >> data1 >> data2;
```

- Can fail if the entered data cannot be interpreted. For example if the variable being read is expecting an integer and what is read is a string of characters the cin will fail and nothing will be assigned to the variable. The cin function will ignore white space and tabs.
- The cin function will only be executed when the enter key is pressed.
- We can use the extraction and insertion operator to work with data from different streams, for example file streams.

Chapter 6

Variables and constants

6.1 What is a variable?

- A variable allows us to use a name for a memory location in RAM in which our variable is stored.
- A variable is an abstraction for a memory location.
- Variables allow programmers to use meaninful names and not memory addresses.
- Variables have:
  - Type: their category (integer, real number, string, objects).
  - Value: the content (10, 3.14, "String").
- Variables must be declared before they are used.
- A variable value may change.
- Example:

  ```
  age = 21; // Compiler error.
  ```

  In the above code we never declared age so the compiler does not know how much memory to allocate for the variable, thus a compiler error is produced, age was never declared. This is called static typing because all the rules are enforced when the program is compiled rather than when the program is running.

  ```
  int age;
  age = 21;
  ```

  The above code is correct.

6.2 Declaring and initializing variables

- The syntax is: `<variable_type> <variable_name>;`.
- Variable types can be anything, such as: `int`, `double`, `float`, `string` you can also declare user defined types (this is object oriented programming) such as `Account`, `Person` with the same syntax you would use with any other type.

6.2.1 Naming rules and conventions

- Naming variables:
  - Can contain letters, numbers and underscores.
  - Must begin with a letter to underscore.
    - Cannot begin with a number digit.
  - Cannot use C++ reserved keywords.
  - Cannot redeclare a name in the same scope.
    - Remember that C++ is case sensative.
  - Example:

Legal	Illegal
Age	int
age	$age
_age	2014_age
My_age	My age
your_age_in_2014	Age+1
INT	cout
Int	return

- The best thing to do is to be consistent with your naming conventions.
  - Stick to the convention you chose from the beggining.
  - Avoid beginning names with underscores.
- Use meaningful names:
  - Not too long and not too short.
- Never use variables before initializing them.
  - Using variables before initializing can cause undefined behaviour.
- Declare variables close to when you need them in your code.

6.2.2 Declaring and initializing

- There are many ways of initializing variables. And all reflect the way C++ has advanced as the years have passed.

```
int age; // uninitialized.
int age = 21; // C-Like initialization.
int age (21); // Constructor initialization.
int age {21}; // C++11 list initialization syntax.
```

- Using the {} list intialization will also check if the assigned values cause an overflow in the program.

6.2.3 Example

```
#include <iostream>
using namespace std;

int main() {
    int room_width {0};
    cout << "Enter the width of the room: ";
    cin >> room_width;
```

```
    cout << "Enter the lenght of the room: ";
    int room_lenght {0};
    cin >> room_lenght;

    cout << "The area is " << room_width*room_lenght << endl;
    return 0;
}
```

6.3 Global variables

- Variables declared outside of any function are called global variables and they can be accessed in any point of your program.
- Since they can be accessed by any part of the program this also means they can be changed by any part of the program which could mean the likelyhood for bugs is higher.
- Local variables are declared and used within a part of the code, as your code progresses out of scope these variables are automatically destoyed.
- Local variables have higher precedence than global variables, if you call a local and global variable the same and decide to use one of both the local is going to be prioritized and the global variable will no be used.

6.4 C++ Built-in Primitive Types

- These are sometimes called fundamental data types because they are implemented directly by the C++ language.
- They include:
  - Character type.
  - Integer type.
    - signed and unsigned.
  - Floating-point types.
  - Boolean types.
- Size and precision is often compiler dependent, this means you must be aware how much storage is allocated for each type. The C++ library `#include <climits>` conatins information about your specific compiler.

6.4.1 Type sizes

- A computer works by storing bits on memory, data types are stored in bits.
- The more bits the more values that can be represented.
- The more bits the more storage requiered.
- The computer however is not conserned with bits but with bytes, a byte is 8 bits stored continously in memory. To find out how many values you can store with n number of bits the formula is 2^n, below are some examples:

Size (in bits)	Representable values	
8	256	2^8
16	65,536	2^{16}
32	4,294,927,296	2^{32}
64	18,446,744,073,551,615	2^{64}

6.4.2 Character types

- Used to represent single characters: 'A', 'X', '@'.
- Wider types are used to represent wide character sets.

Type name	Size / Precision
char	Exactly one byte. At least 8 bits.
char16_t	At least 16 bits.
char32_t	At least 32 bits.
wchar_t	Can represent the largest available character set.

- Example:

```
char m {'j'};
cout << m << endl;
```

6.4.3 Integer data types

- Used to represent whole numbers.
- Signed and unsigned versions.

Type	Size / Precision	Type	Size / Precision
signed short int	At least 16 bits.	unsigned short int	At least 16 bits.
signed int	At least 16 bits.	unsigned int	At least 16 bits.
signed long int	At least 32 bits.	unsigned long int	At least 32 bits.
signed long long int	At least 64 bits.	unsigned long long int	At least 64 bits.

  - Ints can be overflowed, when an overflow happens you can have undefined behaviour.

- Example:

```
unsigned short int exam_score {55};
cout << exam_score << endl;
int countries_represented {65};
cout << countries_represented << endl;
long people_in_florida {2061100000};
cout << people_in_florida << endl;
long long people_on_earth {7'600'000'000};
cout << people_on_earth << endl;
long long distance_to_alpha_century {9'461'000'000'000};
cout << distance_to_alpha_century << endl;
```

6.4.4 Floating point type

- Used to represent non-integer numbers.
- Represented by mantissa and exponent (scientific notation).
- Precision is the number of digits in the mantissa.

- Precission and size are compiler dependent.

Type name	Size / Typical Precision	Typical Range
float	/7 decimal digits	1.2×10^{-38} to 3.4×10^{38}
double	No less than float / 15 decimal digits	2.2×10^{-308} to 1.8×10^{308}
float	No less than double / 19 decimal digits	3.3×10^{-4932} to 1.2×10^{4932}

  - Remember there is no such thing yet as storing an infinitely precise decimal such as π computers store approximations using scientific notation.

- Example:

```
float car_payment {401.23};
cout << car_payment << endl;
double pi {3.14159};
cout << pi << endl;
long double large_amount {2.7e120};
cout << large_amount << endl;
```

6.4.5 Boolean type

- Used to represnt true and false.
- Zero is false.
- Non-zero is true.

Type name	Size / Precision
bool	Usually 8 bits true or false (C++ Keywords)

- Example:

```
bool game_over {false};
cout << game_over << endl;
```

6.4.6 Buffer overflows

```
short val1 {30'000};
short val2 {1'000};
short product {val1 * val2}; // overflow, maximum is about 32,000.
```

6.5 What is the size of a variable?

- The `sizeof` operator determines the size in bytes of a type or variable.
- Examples:

```
sizeof(int);
sizeof(double);
sizeof(some_variable);
sizeof some_variable; // parenthesis are optional for variables.
```

- The sizeof operator gets its information from two C++ include files. `<climits>` `<cfloat>`, the climits and cfloat influde files contain size and precision information about your implemntation of C++. These libraries also provide information constants such as `INT_MAX, INT_MIN, LONG_MAX, LONG_MIN, FLT_MIN, FLT_MAX,`

6.5.1 Examples

*Take into account that the values returned by the sizeof operator will be different depending on your machine.

```cpp
#include <iostream>
#include <climits>
using namespace std;
int main() {
    cout << "sizeof: " << endl;
    cout << "char: " << sizeof(char) << endl;
    cout << "int: " << sizeof(int) << endl;
    cout << "unsigned int: " << sizeof(unsigned int) << endl;
    cout << "short: " << sizeof(short) << endl;
    cout << "long: " << sizeof(long) << endl;
    cout << "long long: " << sizeof(long long) << endl;
    cout << "float: " << sizeof(float) << endl;
    cout << "double: " << sizeof(double) << endl;
    cout << "long double: " << sizeof(long double) << endl;
    cout << "=================<climits>=================" << endl;
    cout << "climits minimum vals" << endl;
    cout << "char: " << CHAR_MIN << endl;
    cout << "int: " << INT_MIN << endl;
    cout << "short: " << SHRT_MIN << endl;
    cout << "long: " << LONG_MIN << endl;
    cout << "long long: " << LLONG_MIN << endl;
    cout << "climits maximum vals" << endl;
    cout << "char: " << CHAR_MAX << endl;
    cout << "int: " << INT_MAX << endl;
    cout << "short: " << SHRT_MAX << endl;
    cout << "long: " << LONG_MAX << endl;
    cout << "long long: " << LLONG_MAX << endl;
    cout << "=================sizeof variable names=================" << endl;
    int age {21};
    cout << "age is " << sizeof(age) << endl;
    double wage {22.24};
    cout << "wage is " << sizeof(wage) << endl;
    return 0;
}
/* OUTPUT:
sizeof:
char: 1
int: 4
unsigned int: 4
short: 2
long: 4
long long: 8
float: 4
double: 8
long double: 16
=================<climits>=================
climits minimum vals
char: -128
int: -2147483648
short: -32768
```

```
long: -2147483648
long long: -9223372036854775808
climits maximum vals
char: 127
int: 2147483647
short: 32767
long: 2147483647
long long: 9223372036854775807
================sizeof variable names=================
age is 4
wage is 8
*/
```

6.6 What is a constant?

- Constants are very much like variables in C++.
- The follow the same naming convensions of variables.
- They occupy storage.
- And they are usually typed.
- However, their value cannot change once declared.

6.6.1 Literal constants

- The most obvious kind of constant.
- Integer literal constants:
  - `12` an integer.
  - `12U` an unsigned integer.
  - `12L` a long integer.
  - `12LL` a long long integer.
- Floating-point literal constants:
  - `12.1` a double.
  - `12.1F` a float.
  - `12.1L` a long double.
- Character literal constants (escape codes).
  - `\n` newline.
  - `\r` return.
  - `\t` tab.
  - `\b` backspace.
  - `\'` single quote.
  - `\"` double quote.
  - `\\` backslash.

6.6.2 Defined constants

- Constants that are declared using the `const` keyword.

```
const double pi {3.1415926};
const int months_in_year {12};
pi = 2.5; // compiler error.
```

Defined constants

- Constants defined as preprocessor directive.

```
#define pi 3.1415926
```

- Notice you don't declare a type, this is sort of a blind find and replace done before compilation of the program, whenever the program encounters `pi` it will replace it with the value of the define constant, since the preprocessor does not know C++ it can't type check and this can make it difficult to find errors. The best is to never define constants and to always use the `const` keyword in your code.

Chapter 7

Arrays and vectors

7.1 Arrays and vectors

- Arrays and vectors are compound data types. Which means they are data types made out of other data types.

7.2 What is an array?

- An array is a compound data type or a data structure.
  - Which means they are data types composed of other data types. They allow us to have collections of elements.
- All elements are of the same type.
- Each element can be accessed directly.
- Why do we need arrays?
  - Suppose we would need to store the scores of a test.
  - We could declare n variables for n students:

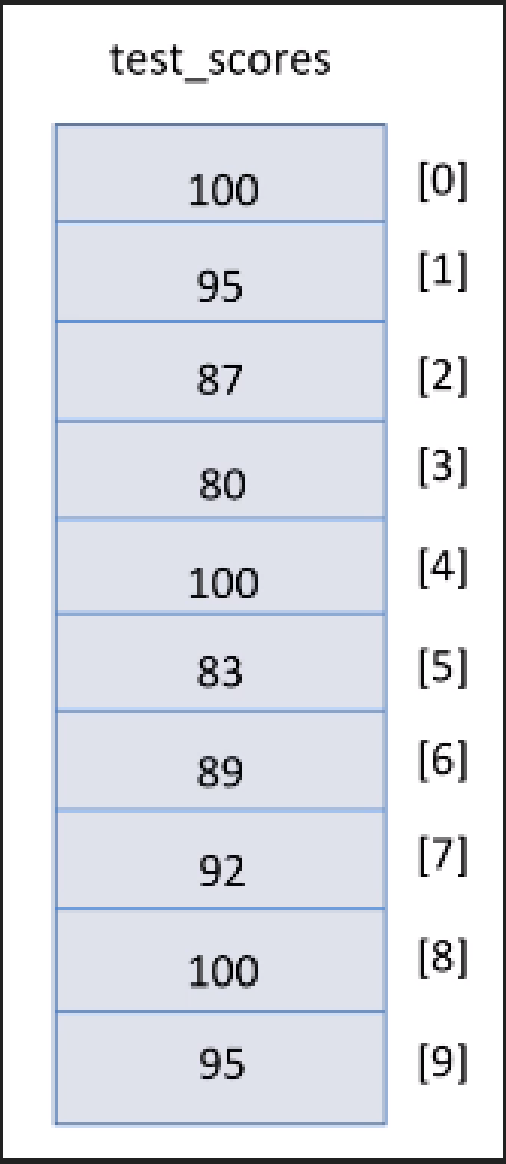
```
int test_score_1 {0};
int test_score_2 {0};
int test_score_3 {0};
int test_score_4 {0};
// ...
int test_score_100 {0};
```

  - This becomes tedious and error prone, now you would need to keep track of 100 variables with their own name. And even worse if I have 1,000,000 test scores this would get out of hand very easily.
  - In this case it's better to use an array.

7.2.1 Characteristics

- Fixed size.
- Elements are all the same type.
- Stored contigously in memory.
- Individual elements can be accessed by their position or index.

- First element is at index 0.
- Last element is at index $size - 1$.
- No checking to see if you are out of bounds. It is the responsability of the programmer to check if the program is writing data out of bounds, this can cause undefined behaviour.
- Always initialize arrays.
- Very efficient.
- Iteration (looping) is often used to process the elements stored.



7.3 Declaring and initializing arrays

7.3.1 Declaring arrays

- `element_type array_name [constant number of elements];`

  - The constant number of elements can be an expression that evaluates to a constant, a constant, a number or anything that results in a number before compiling.

```
int test_scores[5];
int high_score_per_level[10];
const double days_in_year {365};
double hi_temperature[days_in_year];
```

7.3.2 Initializing arrays

- `element_type array_name [number of elements] {init elements};`

```
int test_scores[5] {100,95,99,87,88};
int high_score_per_level[10] {3,5}; // init to 3,5 and remaining to 0.
const double days_in_year {365};
double hi_temperatures[days_in_year] {0}; // init all to zero.
double hi_temperature[days_in_year] {1,2,3,4,5}; // size automatically calculated.
```

7.4 Accessing and modifying array elements

- Accessing array elements: `array_name [element_index];`.

```
#include <iostream>
using namespace std;
int main() {
    int test_scores[5] {100,95,99,87,88};
    cout << "index 0: " << test_scores[0] << endl;
    cout << "index 1: " << test_scores[1] << endl;
    cout << "index 2: " << test_scores[2] << endl;
    cout << "index 3: " << test_scores[3] << endl;
    cout << "index 4: " << test_scores[4] << endl;
    return 0;
}
/* OUTPUT:
index 0: 100
index 1: 95
index 2: 99
index 3: 87
index 4: 88
*/
```

- Changing contents of array elements:

```
#include <iostream>
using namespace std;
int main() {
    int test_scores[5] {0};
    cin >> test_scores[0];
    cin >> test_scores[1];
    cin >> test_scores[2];
    cin >> test_scores[3];
    cin >> test_scores[4];
    test_scores[0] = 90;
    return 0;
}
```

7.4.1 How do arrays work?

- The nane of the array represents the location of the first element in the array (index 0).
- The [index] represents the offset from the beggining of the array.
- C++ simply performs a calculation to find the correct element.
- There is no bounds checking.

7.4.2 Examples

```
#include <iostream>
using namespace std;
int main() {
    char vowels[] {'a','e','i','o','u'};
    cout << vowels[0] << endl;
    cout << vowels[4] << endl;
    cin >> vowels[5]; // out of bounds.
    return 0;
}
/* OUTPUT:
    a
    u
    ... program crashes.
*/

#include <iostream>
using namespace std;
int main() {
    double hi_temps[] {90.1, 89.8, 77.5, 81.6};
    cout << "\nThe fist high temperature is: " << hi_temps[0] << endl;
    hi_temps[0] = 100.7; // sets the first element of hi_temps to 100.7.
    cout << "\nThe fist is now: " << hi_temps[0] << endl;
    return 0;
}
/* OUTPUT:

The fist high temperature is: 90.1

The fist is now: 100.7

*/

#include <iostream>
using namespace std;
int main() {
    int test_scores[5] {100};
    cout << "\nFirst score at index 0: " << test_scores[0] << endl;
    cout << "\nSecond score at index 1: " << test_scores[1] << endl;
    cout << "\nThird score at index 2: " << test_scores[2] << endl;
    cout << "\nFourth score at index 3: " << test_scores[3] << endl;
    cout << "\nFifth score at index 4: " << test_scores[4] << endl;
    cin >> test_scores[0];
    cin >> test_scores[1];
    cin >> test_scores[2];
```

```
    cin >> test_scores[3];
    cin >> test_scores[4];
    cout << "\nFirst score at index 0: " << test_scores[0] << endl;
    cout << "\nSecond score at index 1: " << test_scores[1] << endl;
    cout << "\nThird score at index 2: " << test_scores[2] << endl;
    cout << "\nFourth score at index 3: " << test_scores[3] << endl;
    cout << "\nFifth score at index 4: " << test_scores[4] << endl;
    return 0;
}
/* OUTPUT:
First score at index 0: 100

Second score at index 1: 0

Third score at index 2: 0

Fourth score at index 3: 0

Fifth score at index 4:

0 1 2 3 4 5

First score at index 0: 1

Second score at index 1: 2

Third score at index 2: 3

Fourth score at index 3: 4

Fifth score at index 4: 5
*/

#include <iostream>
using namespace std;
int main() {
    int test_scores[5] {100};
    cout << "Name of array: " << test_scores << endl;
    return 0;
}
/* OUTPUT:
Name of array: 0x61fe00
*/
```

7.5 Multidimentional arrays

- Declaring multidimentional arrays: `element_type array_name [dimension 1 size][dimension 2 size]...;`

  `int movie_rating [3][4];`

- Some compilers do place limits on the number of dimensions you have depending on what machine they are running and the compiler being used.
- Multi-dimentional arrays are a spreadsheet concept, think of the first dimension as the rows and the second as the columns.

```
int spread_sheet[][] {
    {1,2,3},
    {4,5,6},
    {7,8,9}
};
```

7.6 Declaring and initializing vectors

- Suppose we want to store test scores for my school.
- I have no way of knowing how many students will register next year.
- Options:
  - Pick a size that you are not likely to exceed and use static arrays.
  - Use a dynamic array sich as a vector.

7.6.1 Vectors

- A C++ vector is a container in the C++ standard template library.
- An array that can grom and shrink in size at excecution time.
- Provides similar semantics and syntax as arrays.
- Very efficient.
- Can provide bounds checking.
- Can use lots of cool functions like sort, reverse, find and more.
- When we create a C++ vector we are creating a C++ object and this means it can perform operations for us.

7.6.2 Declaring vectors

- We must include the vector library and the standard library, both allow us to create C++ vectors.
- Syntax: `vector<datatype> vector_name;`.

```
#include <vector>
using namespace std;
vector<char> vowels;
vector<int> test_scores;
```

7.6.3 Initializing vectors

- For vectors, since they are objects, we can provide initialization with the constructor intialization style:

```
vector<char> vowels (5);
vector<int> test_scores (10);
```

  - The above code creates a vector of size 5 if chars, then a vector of size 10 of ints, also all the elements of the array are intialized automatically to zero, you no longer need to do that explicitly.

- We can also use the curly brace initialization style to specify the value of the elements we want:

```
vector<char> vowels {'a','e','i','o','u'};
// Initializes vowels to the specified values in a vector of size 5.
vector<int> test_scores {100,98,89,85,93};
// Initializes test_scores to the 5 specified values.
vector<double> hi_temperatures (365, 80.0);
// The first paramenter is the initial size of the vector which os 365, the second is to which valu
```

7.6.4 Vector caracteristics

- Dynamic size.
- Elements are all the same type.
- Stored contiguosly in memory.
- Individual elements can be accessed by their position or index.
- First element is at index 0.
- Last element is at index $size - 1$.
- If you use the subscript operator ([index]) then vectors will provide no bounds checking, however the vector object allows you to get information at an index using methods that do provide bounds checking.
- Provides many useful functions that do bounds checking.
- Elements are automatically initialized to zero unless you specify otherwise.
- Very efficient.
- Iteration (looping) is ften used to process the elements.

7.7 Accessing and modifying vector elements

- The first way to access vector elements is to use array syntax (with the subscript operator): `vector_name[element_inde` with this way no bounds checking will be done.

```
vector<int> test_scores {100,95,99,87,88};
cout << test_scores[0] << endl;
cout << test_scores[1] << endl;
cout << test_scores[2] << endl;
cout << test_scores[3] << endl;
cout << test_scores[4] << endl;
```

- We can also access vector elements with the `.at` method, the syntax is: `vector_name.at(element_index);`

```
vector<int> test_scores {100,95,99,87,88};
cout << test_scores.at(0) << endl;
cout << test_scores.at(1) << endl;
cout << test_scores.at(2) << endl;
cout << test_scores.at(3) << endl;
cout << test_scores.at(4) << endl;
```

7.7.1 Vectors changing in size dynamically

- The vector has a method called `push_back` that adds a new element (of the same type) to the back of the vector.
- This vector method will automatically allocate the required space.

7.7.2 Out of bounds errors

- What if you are out of bounds?
- Arrays never do bounds checking.
- Many vector methods provide bounds checking.
- An exception and error message is generated.

Chapter 8

Statements and operators

8.1 Expressions and statements

- An expression is:
  - The most basic building block of a program.
  - "A sequence of operatiors and operands that specifies a computation." — C++ STD.
  - Computes a value from a number of operands.
  - There is much, much more to expressions — Not necessary at this level.
- Examples of expressions:

```
34 // literal.
favorite_number // variable.
1.5 + 2.8 // addition.
2 * 5 // multiplication.
a > b // relational.
a = b // assignment.
```

- Difference between a statement and expression:
  - A statement is:
    - A complete line of code that performs some action.
    - Usually terminated with a semi-colon.
    - Usually contain expressions.
    - C++ has many types of statements:
      - Expression, null, compound, selection, iteration, declaration, jump, try blocks, and others.
    - Expressions are used to make up the statement.

```
// Example of statements:
int x; // declaration.
favorite_number = 12; // assignment.
1.5 + 2.8; // expression.
x = 2 * 5; // assignment.
if ( a > b ) cout << "a is greater than b"; // if statement.
```

  - A null statement is a statement that doesn't perform any computation, it would be the equivalent of the following example:

```
int a;
for (int i = 0; i < 10; i ++ ) {
    ; // null statement.
}
```

8.2 Using operators

- C++ has a rich set of operators, most of them are binary operators which means they operate on two operands, for example the multiplication operator operates on two operands (numbers). However C++'s operators aren't just binary.
- C++ has a rich set of operators.
  - Unary (operates on one operand), binary (operates on two operands), ternary (operates on three operands).
- Common operators can be grouped as follows:
  - Assignment: used for modifying the value of some object by assigning a new value to it.
  - Arithmetic: used to perform mathematical operations on operands.
  - Increment/Decrement: these operators work as an assignment and arithmetic, they increment or decrement the operand by one.
  - Relational/Comparison: allow you to compare the value of objects, examples include: $=, \neq, \geq, \leq$, etcetera.
  - Logical: used to test for logical or boolean conditions, for example: if you want to execute certain part of your code only when the temperature is less than freezing. These include the logical not, and, or operators.
  - Member access: used to allow access to a specific member. An example is the array subscript operator ([]). others that work with objects and pointers which will be introduced later.
  - Other: other operators.

8.3 The assignment operator

- Nearly all programming languages have the ability to change the value of a variable.
- In C++ we can change the value stored in a variable using the assignment operator (=).

  ```
  lhs = rhs;
  ```

- This does not represent equality, this represents that the value of `rhs` will be the value of `lhs`.
- We are not asserting that the left-hand side is equal to the right-hand side, nor are we comparing the left-hand side to the right-hand side. We are evaluating the value of the expression on the right-hand side and storing the value to the left-hand sign.
- `rhs` is an expression that is evaluated to a value.
- The value of the `rhs` is stored to the `lhs`.
- The value of the `rsh` must be type compatible with the `lhs`, meaning they must be of the same type.
- C++ is statcally typed which means that the compiler will be checking if it makes sense to asssign the right-hand side to the left-hand side, if it doesn't make sense you'll get a compiler error.
- In order to store the right-hand side to the left-hand side, the left-hand side must be assignable, it can't be a literal, it can't be a constant.
- An assignment expression is evaluated to what was just assigned, this makes it posible to chain more than one variable in a single assignment, such as:

  ```
  a = b = c = d = e = 10;
  ```

- It is important to understand that assignment is not initialization, initialization happens only when the variable is declared and the variable gets that value for the very first time, assignment is when you change a value that already exists in the variable after initializing it.

8.3.1 Example

- Example:
  - In C++ there is a concept in assignment know as r-value and l-value, whenever we say the r-value we are denoting the content of some variable or an expression that evaluates to a value, the l-value is the a location, the statement: `lhs=rhs` means "store whatever value is in `rhs` to location `lhs`". The `rhs` could be very complex, the program will evaluate and compute that value and store it in location `lhs`.
  - In C++ all this checking is done at compile time, thus when something compiles without errors or warnings you are guaranteed to have done it correctly. This is also because C++ is statically typed.

```cpp
#include <iostream>
using namespace std;
int main() {
    int num1 {10}; // initialization.
    int num2 {20}; // initialization.
    num1 = 100; // assignment. lhs = rhs;
    cout << "num1 is " << num1 << endl;
    cout << "num2 is " << num2 << endl;
    cout << endl;
    return 0;
}
/* OUTPUT:
num1 is 100
num2 is 20

*/
```

- Chain assignment:
  - In assignment operators they are done from right to left, the left-hand side of the expression is done last, while the further right part of the expression is done first.
  - So something like this:

$$num1 = num2 = num3 = 1'000;$$
$$num1 = num2 = \underbrace{num3 = 1'000;}_{\text{Evaluates to 1'000}}$$
$$num1 = \underbrace{num2 = num3 = 1'000;}_{\text{Evaluates to 1'000}}$$

    - Finally the last part of the assignment is assigning the value of num2 to num1.

```cpp
#include <iostream>
using namespace std;
int main() {
    int num1 {10}; // initialization.
    int num2 {20}; // initialization.
    num1 = num2 = 1'000; // chained assignment.
    cout << "num1 is " << num1 << endl;
    cout << "num2 is " << num2 << endl;
    cout << endl;
    return 0;
```

```
}
/* OUTPUT:
num1 is 1000
num2 is 1000

*/
```

- The compiler will constantly be checking if it makes sense to assign something to a variable. For example the following code produces a compiler error because an `int` variable can only hold integers not strings.

```
#include <iostream>
using namespace std;
int main() {
    int num1 {10}; // initialization.
    int num2 {20}; // initialization.
    num1 = "String"; // assignment to a string of an integer variable is illegal.
    cout << "num1 is " << num1 << endl;
    cout << "num2 is " << num2 << endl;
    cout << endl;
    return 0;
}
/* OUTPUT: Compiler error.
./Code/main.cpp:6:12: error: invalid conversion from 'const char*' to 'int' [-fpermissive]
    num1 = "String"; // assignment.
*/
```

- The following code will produce an error, though we are assigning an int to the variable which makes sense, we are declaring it as a constant, thus we cannot change the value of that constant after initialization.

```
#include <iostream>
using namespace std;
int main() {
    int const num1 {10}; // initialization.
    int num2 {20}; // initialization.
    num1 = 100; // assignment to a constant is illegal.
    cout << "num1 is " << num1 << endl;
    cout << "num2 is " << num2 << endl;
    cout << endl;
    return 0;
}
/* OUTPUT: Compiler error.
./Code/main.cpp:6:12: error: assignment of read-only variable 'num1'
    num1 = 100; // assignment to a constant is illegal.
*/
```

- The following code will produce a compiler error because we are assigning a variable to a literal:

```
#include <iostream>
using namespace std;
int main() {
    int num1 {10};
    100 = num1; // assignment
    return 0;
}
```

```
/* OUTPUT: Compiler error.
./Code/main.cpp:5:11: error: lvalue required as left operand of assignment
    100 = num1; // 100 is a literal and does not have a location in memory.
*/
```

8.4 Arithmetic operators

- The arithmetic operators are:
  - Addition: $+$
  - Subtraction: $-$
  - Multiplication: $*$
  - Division: $/$
  - Modulo or remainder (works only with integers): $\%$
- These operators can be overloaded, what 'overloaded' means is that they work with different types, for example you can use the addition operator to add floats, integers, doubles, etcetera.

8.4.1 Examples

- Adding variables and assigning the result to another.

```
#include <iostream>
using namespace std;
int main() {
    int num1 {100};
    int num2 {200};
    int result {num1 + num2};
    cout << num1 << " + " << num2 << " = " << result << endl;
    return 0;
}
/* OUTPUT:
100 + 200 = 300
*/
```

- Arithmetic:
  - It is important to notice that multiplication is not like algebra where you can omit adding the multiplication symbol, in C++ we need to add the multiplication symbol explicitly. There are no such thing as: (3)(4) rather write it as: `(3)*(4);`.
  - Another thing to watch out for is when dividing integers you don't get a decimal part, you will only get the whole number part, dividing $100/200 = 0.5$ however we just store the whole part, which is to say as far as C++ is concerned when you divide the integers 100/200 the result is floor division and you are left with the result being 0.
  - The modulus operator is the remainder of division, for example if we divide 10 by 3 we get that we can divide 10 by 3 a total of 3 times, however we are left with a remainder, that remainder is 1 ($3 \times 3 + 1 = 10$).
  - There is operator precedence in C++, it goes in the order of PEMDAS, P(Parenthesis), E(Exponents), M(multiplication), D(division), A(addition), and S(subtraction). So for example in the expression $(8 * 10 + 1) - 10$ the parenthesis are evaluated first, inside the parenthesis the multiplication of 8 and 10 are evaluated and the one is added resulting in 81, then the subtraction of 10 is done and the result is 71.

```cpp
#include <iostream>
using namespace std;
int main() {
    int num1 {100};
    int num2 {200};
    int result {0};
    result = num1 + num2;
    cout << num1 << " + " << num2 << " = " << result << endl;
    result = num1 - num2;
    cout << num1 << " - " << num2 << " = " << result << endl;
    result = num1 * num2;
    cout << num1 << " * " << num2 << " = " << result << endl;
    result = num1 / num2; // floor division
    cout << num1 << " / " << num2 << " = " << result << endl;
    result = num1 % num2;
    cout << num1 << " % " << num2 << " = " << result << endl;
    return 0;
}
/* OUTPUT:
100 + 200 = 300
100 - 200 = -100
100 * 200 = 20000
100 / 200 = 0
100 % 200 = 100

*/
```

8.5 Increment and decrement operators

- The increment and decrement operators are unary operators, `++`, `--` the `++`.
- The increment and decrement operators are simply just saying: `++` increment its operand by one, `--` decrement the operand by one.
- This operator can also be overloaded, which is to say that they will increment or decrement different types, such as incrementing or decrementing floats, doubles, integers, and even pointers.
- There are two variants to this operator, postfix and suffix.
  - Postfix notation: `++num` this means to increment num by one before using it.
  - Prefix notation: `num++` this means to increment num by one after using it.
- Don't overuse this operator, **never use it twice for the same variable in the same statement** because that will cause undefined behavior. Things such as the following are not allowed:

```cpp
num = num1++ + ++num1; // or
cout << i++ << ++i << i << endl; // will cause undefined bahaviour.
```

- The post fix and prefix notation work exactly the same if they are alone on one line, such as: `num++;`. However, if they are accompanied such as: `arr[num++] = 10;` will work differently.

8.5.1 Example

- Incrementing:

```cpp
#include <iostream>
using namespace std;
int main() {
    int counter {10};
    int result {0};
    cout << "Counter: " << counter << endl;
    counter = counter + 1; // 11
    cout << "Counter: " << counter << endl;
    counter++;
    cout << "Counter: " << counter << endl;
    ++counter;
    cout << "Counter: " << counter << endl;
    return 0;
}
/* OUTPUT:
Counter: 10
Counter: 11
Counter: 12
Counter: 13

*/
```

- Pre-increment example:

```cpp
#include <iostream>
using namespace std;
int main() {
    int counter {10};
    int result {0};
    result = ++counter; // Counter will be incremented before its used.
    cout << "Result: " << result << endl;
    cout << "Counter: " << counter << endl;
    return 0;
}
/* OUTPUT:
Result: 11
Counter: 11
*/
```

- Post-increment example:

```cpp
#include <iostream>
using namespace std;
int main() {
    int counter {10};
    int result {0};
    result = counter++; // Counter will be incremented after its used.
    cout << "Result: " << result << endl;
    cout << "Counter: " << counter << endl;
    return 0;
}
/* OUTPUT:
Result: 10
Counter: 11
*/
```

- Saying `i = ++num + 10;` is the same as saying:

$$Consideringnum = 10.$$
$$i = \underbrace{++num}_{num=num+1 \rightarrow num=11} + 10;$$
$$i = \underbrace{num}_{\text{has been incremented by one}} + 10;$$
$$i = 11 + 10 = 21;$$

```
#include <iostream>
using namespace std;
int main() {
    int num {10};
    int i {0};
    i = ++num + 10;
    cout << "i: " << i << endl;
    return 0;
}
/* OUTPUT:
i: 21
*/
```

- The same example with the post-increment:

```
#include <iostream>
using namespace std;
int main() {
    int num {10};
    int i {0};
    i = num++ + 10; // for this addition it will use the value of num as is and then increment so\a
    cout << "i: " << i << endl;
    return 0;
}
/* OUTPUT:
i: 20
*/
```

8.6 Mixed expressions and conversions

- C++ operations occur on the same type operands. For example $a + b$ where a is an integer and b is a double.
- If operands are of different types, C++ will convert one. In many cases this happens automatically.
- This is important since it could affect calculation results.
- C++ will attempt to automatically convert types (coercion). If automatic conversion or coercion is not possible a compiler error will occur.

8.6.1 Conversions

- In order to understand how these automatic conversions happen we need to understand higher vs lower types.

  - Higher type is a type that can hold higher or more values.
  - Lower type is a type that can hold lower or less values than the higher type.
- Higher vs Lower types are based on the size of the values the type can hold.
  - `long double`, `double`, `float`, `unsigned long`, `long`, `unsigned int`, `int`, `short int`, `char` are always converted to an `int`.
- Coercion always happens by converting the lower type to the higher type, because the lower type will always fit in to the higher type whereas the higher type will almost never fit in to the lower type.
- Type coercion: happens when we convert a lower type operand to a higher type operand in order to operate them. For example, adding an integer and a double, automatic conversion will occur, first the integer will get converted to a double, then added.
- Promotion: conversion to a higher type:
  - Used in mathematical expressions.
- Demotion: conversion to a lower type:
  - Used with assignment to lower types.
  - For example when performing integer division the result is operated on a double and then the decimal part is truncated until we are left with just the whole number part.

8.6.2 Example type coercion

- Lower operand to higher operand, the lower is promoted to a higher.

```
2 * 2.5; // 2 is promoted to 2.0, integer -> double.
```

- lower = higher; the higher is demoted to a lower: (this risks potentially loosing information)

```
int num {0};
num = 100.2; // float demoted to a int.
```

8.6.3 Examples explicit type casting

- We can explicitly cast or coerce to a certain type.

```
#include <iostream>
using namespace std;
int main() {
    int total_amount {100};
    int total_number {8};
    double average {0.0};
    average = total_amount / total_number;
    cout << "Without explicit coercion: " << average << endl; // Here we are doing integer divisior
    average = static_cast<double>(total_amount) / total_number; // now one of the operands is a dou
    cout << "With explicit coersion: " << average << endl;
    return 0;
}
/* OUTPUT:
Without explicit coercion: 12
With explicit coersion: 12.5
*/
```

- There are two major ways of casting, the first is the syntax we already have seen: `static_cast<double>(var1);` the second is the c-style cast which is `(double)var1;`. It is better to use the C++ style cast since it is more robust as it also checks for congruence and if it makes sense to cast the variable, rather than the C-style cast just does it and it doesn't check for congruence nor if it makes sense.

```
#include <iostream>
using namespace std;
int main() {
    int total {};
    int num1 {}, num2 {}, num3 {};
    const int count {3};
    cout << "Enter 3 integers separated by spaces: ";
    cin >> num1 >> num2 >> num3;
    cout << endl;
    total = num1 + num2 + num3;
    double average {0.0};
    average = static_cast<double>(total) / count; // C++ style cast.
    // average = (double)(total) / count; // older C-style cast
    cout << average << endl;
    return 0;
}
/* OUTPUT:
Enter 3 integers separated by spaces: 15 13 12

13.3333

*/
```

8.7 Testing for equality

- These operators compare the values of two expressions and evaluate to a boolean.
- These operators are the `==` which is the equality operator and the `!=` which is the not equals operator.

```
expr1 == expr2; // evaluates to true if both values are the same, else false.
expr1 != expr2; // evaluates to false if both values are the same, else true.
100 == 200; // evaluates to false.
num1 != num2; // evaluates to true if they are different, else false.
```

- Remember to use two equals signs not just one.
- Another example:

```
bool result {false};
result = (100 == 50+50); // store the boolean in result.
```

- In C++ there is a standard library string manipulator called `std::boolalpha` once you use it all boolean output printed to console will result in the words "true" or "false" being printed instead of 0 or 1. `std::noboolalpha` will deactivate this feature and allows you to print the default of 0 and 1.

```
#include <iostream>
using namespace std;
int main() {
    int num1 {10}, num2 {10};
    cout << "num1: " << num1 << ", num2:" << num2 << endl;
    cout << "(num1 == num2) -> " << (num1 == num2) << endl; // 0 or 1
```

```
    cout << "(num1 != num2) -> " << (num1 != num2) << endl;
    cout << std::boolalpha;
    cout << "(num1 == num2) -> " << (num1 == num2) << endl; // true or false
    cout << "(num1 != num2) -> " << (num1 != num2) << endl;
    cout << std::noboolalpha;
    return 0;
}
/* OUTPUT:
num1: 10, num2:10
(num1 == num2) -> 1
(num1 != num2) -> 0
(num1 == num2) -> true
(num1 != num2) -> false
*/
```

8.7.1 Example

- Example with integers: keep in mind you can do this with any primitive type and with the correct overloading with any user defined type.

```
#include <iostream>
using namespace std;

int main() {
    cout << boolalpha;
    int num1 {0}, num2 {0};
    bool equal_result {false};
    bool not_equal_result;
    cout << "Enter 2 integers: " << endl;
    cin >> num1 >> num2;
    equal_result = (num1 == num2);
    not_equal_result = (num1 != num2);
    cout << "Comparison result (equals): " << equal_result << endl;
    cout << "Comparison result (not equals): " << not_equal_result << endl;
    return 0;
}
/* OUTPUT:
Enter 2 integers:
10 10
Comparison result (equals):true
Comparison result (not equals): false
*/
```

- Example with characters:

```
#include <iostream>
using namespace std;

int main() {
    cout << boolalpha;
    bool equal_result {false}, not_equal_result {false};
    char char1{}, char2{};
    cout << "Enter two characters separated by a space: ";
    cin >> char1 >> char2;
    equal_result = (char1 == char2);
```

```
    not_equal_result = (char1 != char2);
    cout << "Comparision result (equals) : " << equal_result << endl;
    cout << "Comparision result (not equals) : " << not_equal_result << endl;
    return 0;
}
/* OUTPUT:
Enter two characters separated by a space: a a
Comparision result (equals) : true
Comparision result (not equals) : false
*/
```

- Example with doubles:
  - The following evaluates to true because of the way computers store information, as far as it is concerned 11.9999999999999999 is 12.

```
#include <iostream>
using namespace std;

int main() {
    cout << boolalpha;
    bool equal_result {false}, not_equal_result {false};
    double double1{}, double2{};
    cout << "Enter two doubles separated by a space: ";
    cin >> double1 >> double2;
    equal_result = (double1 == double2);
    not_equal_result = (double1 != double2);
    cout << "Comparision result (equals) : " << equal_result << endl;
    cout << "Comparision result (not equals) : " << not_equal_result << endl;
    return 0;
}
/* OUTPUT:
Enter two doubles separated by a space: 12 11.9999999999999999
Comparision result (equals) : true
Comparision result (not equals) : false
*/
```

- Example with integer and a double:
  - In the below example we see another example of coercion, the 10 will get promoted to 10.0 then compared with the other with is 10.0, the expression evaluates to true.

```
#include <iostream>
using namespace std;

int main() {
    cout << boolalpha;
    bool equal_result {false}, not_equal_result {false};
    int num1 {};
    double double1 {};
    cout << "Enter an integer and a double separated by a space: ";
    cin >> num1 >> double1;
    equal_result = (num1 == double1);
    not_equal_result = (num1 != double1);
    cout << "Comparision result (equals) : " << equal_result << endl;
    cout << "Comparision result (not equals) : " << not_equal_result << endl;
```

```
    return 0;
}
/* OUTPUT:
Enter an integer and a double separated by a space: 10 10.0
Comparision result (equals) : true
Comparision result (not equals) : false
*/
```

8.8 Relational operators

- In addition to the equality operators C++ also provides another set of operators to compare objects.

Operator	Meaning
>	greater than.
>=	greater than or equal to.
<	less than.
<=	less than or equal to.
<=>	three-way comparison (C++20).

  - The three-way comparison operator compares two expressions and evaluates to 0 if they are equal, less than 0 if the left-hand side is greater than the right-hand side, and greater than 0 if the right-hand side is greater than the left-hand side.

8.8.1 Example

- Example:

```
#include <iostream>
using namespace std;

int main() {
    cout << boolalpha;
    int num1 {}, num2 {};
    cout << "Enter 2 integers separated by a space: ";
    cin >> num1 >> num2;
    cout << num1 << " > "   << num2  << " : " << (num1 > num2) << endl;
    cout << num1 << " >= " << num2  << " : " << (num1 >= num2) << endl;
    cout << num1 << " < "  << num2  << " : " << (num1 < num2) << endl;
    cout << num1 << " <= " << num2  << " : " << (num1 <= num2) << endl;
    cout << endl;
    return 0;
}
/* OUTPUT:
Enter 2 integers separated by a space: 10 20
10 > 20 : false
10 >= 20 : false
10 < 20 : true
10 <= 20 : true

*/
```

- Checking user obedience:

```
#include <iostream>
using namespace std;
```

```cpp
int main() {
    cout << boolalpha;
    int num1 {}, num2 {};
    const int lower {10};
    const int upper {20};
    cout << "Enter an integer that is greater than " << lower << " : " ;
    cin >> num1;
    cout << num1 << " > " << lower << " is " << (num1 > lower) << endl;
    cout << "Enter an integer that is less than or equal to " << upper << " : " ;
    cin >> num1;
    cout << num1 << " <= " << upper << " is " << (num1 <=  upper) << endl;
    cout  << endl;
    return 0;
}
/* OUTPUT:
Enter an integer that is greater than 10 : 12
12 > 10 is true
Enter an integer that is less than or equal to 20 : 16
16 <= 20 is true

*/
```

8.9 Logical operators

- C++ has three logical operators:

Operator	Meaning
`!`	negation, logical not.
`&&`	logical and.
`||`	logical or.

- These operators work on boolean expressions and evaluate to boolean values themselves.
- There are two ways to write the logical operators, you can write the keywords: `not`, `and`, `or` or the symbols `!`, `&&`, `||`.
- The `!` operator is a unary operator and it simply negates the value.

expression a	not a / !a
true	false
false	true

- The `&&` operator is a binary operator and it performs a logical and.

Expression a	Expression b	a and b / a && b
true	true	true
true	false	false
false	true	false
false	false	false

- The `||` operator is a binary operator and it performs the logical or.

Expression a	Expression b	a or b / a `||` b
true	true	true
true	false	true
false	true	true
false	false	false

8.9.1 Precedence

- Logical operators have precedence.
- The `!` or logical *not* has higher precedence than the logical *and*.
- The logical *and* has higher precedence than the logical *or*.
- The logical *not* is a unary operator.
- The logical *and* and logical *or* are binary operators.

8.9.2 Examples

- In the below code we write the statements `10 <= 20 && 20 <= 30`, we cannot write chained logical operators such as for example: $10 \leq 20 \leq 30$ we would need to write $10 \leq 20$ and $20 \leq 30$.
- Other examples:

```
num1 >= 10 && num1 < 20; // return true if num1 is greater or equal to 10 and num1 is less than 20
num1 <= 10 || num1 >= 20; // return true if num1 is less than or equal to 10 or num1 greater or equ
!is_raining && temperature > 32.0; // return true if it is not raining and the temperature is above
is_raining || is_snowing; // returns true if it is raining or snowing.
temperature > 100 && is_humid || is_raining; // returns true if temperature is above 100 and it is
```

8.9.3 Short-Circuit evaluation

- When evaluating a logical expression in C++ stops as soon as the result is known, for example:

```
expr1 && expr2 && expr3; // if expr1 is false there is no way the statement is true, so it simply s
expr1 || expr2 || expr3; // if expr1 is true it already knows the statement is true because only or
```

8.9.4 Example

- Example of user entering numbers within bounds:

```
#include <iostream>
using namespace std;
int main() {
    int num {0};
    const int lower {10}, upper {20};
    cout << boolalpha;
    cout << "Enter an integer - the bounds are " << lower << " and " << upper << ": ";
    cin >> num;
    bool within_bounds {false};
    within_bounds = (num > lower && num < upper);
    cout << num << " is between " << lower << " and " << upper << " : " << within_bounds << endl;
    return 0;
}
/* OUTPUT:
Enter an integer - the bounds are 10 and 20: 15 13
```

```
15 is between 10 and 20 : true

*/
```

- Example outside the lower and upper bounds:

```
#include <iostream>
using namespace std;
int main() {
    int num {0};
    const int lower {10}, upper {20};
    cout << boolalpha;
    cout << "Enter an integer out of bounds - the bounds are " << lower << " and " << upper << ": "
    cin >> num;
    bool within_bounds {false};
    within_bounds = (num < lower || num > upper);
    cout << num << " is outside " << lower << " and " << upper << " : " << within_bounds << endl;
    return 0;
}
/* OUTPUT:
Enter an integer out of bounds - the bounds are 10 and 20: 3 45
3 is outside 10 and 20 : true

*/
```

8.10 Compound assignment operators

- Compound assignment operators:

Operator	Example	Meaning
+=	lhs += rhs;	lhs = lhs + (rhs);
-=	lhs -= rhs;	lhs = lhs - (rhs);
*=	lhs *= rhs;	lhs = lhs * (rhs);
/=	lhs /= rhs;	lhs = lhs / (rhs);
%=	lhs %= rhs;	lhs = lhs % (rhs);
>>=	lhs >>= rhs;	lhs = lhs >> (rhs);
<<=	lhs <<= rhs;	lhs = lhs << (rhs);
&=	lhs &= rhs;	lhs = lhs & (rhs);
^=	lhs ^= rhs;	lhs = lhs ^ (rhs);
|=	lhs |= rhs;	lhs = lhs | (rhs);

- Example:

```
a += 1; // a = a + 1
a /= 5; // a = a / 5
a *= b + c; // a = a * (b + c)
```

8.11 Operator Precedence

You can find a table of the complete precedence and associativity here.

- What is associativity?
  - Use precedence rules when adjacent operators are different.

expr1 **op1** expr2 **op2** expr3 // precedence.

- Use associativity rules when adjacent operators have the same precedence.

expr1 **op1** expr2 **op1** expr3 // associativity.

- Use parenthesis to absolutely remove any doubt.

8.11.1 Example

- Precedence viewed with parenthesis:

$$result = num1 + num2 * num3;$$
$$result = (num1 + (num2 * num3));$$

- Use associativity from left to right since additiona and subtraction have the same precedence.

$$result = num1 + num2 - num3;$$
$$result = ((num1 + num2) - num3);$$

Chapter 9

Controlling program flow

9.1 Section overview

- Sequence: So far what we've learned has been sequential programming, a series of statements running sequentially, our code can't make decisions.
- Selection structures: allow us to make decisions based on certain conditions being true or false.
- Iteration: looping or repeating.

With sequence, selection and iteration we can implement any algorithm.

9.1.1 Selection: Decision

- `if` statement.
- `if else` statement.
- Nested `if` statements.
- `switch` statement.
- Conditional operator `?:`;

9.1.2 Iteration: Looping

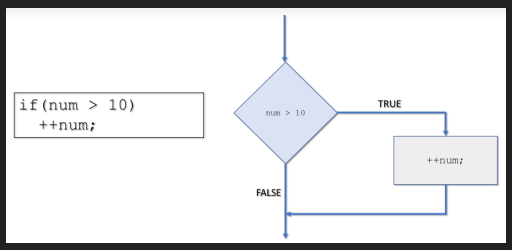
- `for` loop.
- Range-Based `for` loop.
- `while` loop.
- `continue` and `break`.
- Infinite loops.
- Nested loops.

9.2 If statement

- Syntax is: `if (expr) { statement; }`.
- The control expression must evaluate to a boolean true or false value.
- If the expression is true then execute the statement.
- If the expression is false then skip the statement.
- As a recommendation we always indent the code inside the if statement.

9.2.1 Examples of single if statements

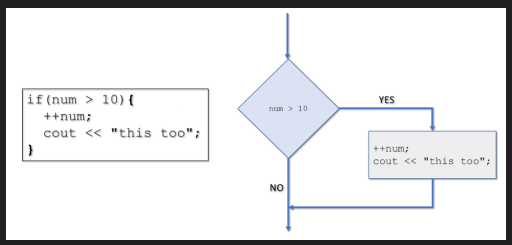
For single if statements you do not need to enclose the statements in curly braces.




```
if (selection == 'A')
    cout << "You selected A";
if (num > 10)
    cout << "num is grater than 10";
if (health < 100 && player_healed)
    health = 100;
```

9.2.2 Block if statements

- A block of code we have already seen in the main function.
- Create a block of code by including more than one statement in code block and adding {}.
- Blocks can also contain variable declarations.
- These variables are visible only within the block (local scope).



9.2.3 Example of block ifs

```
#include <iostream>
using namespace std;
int main() {
    int num {0};
```

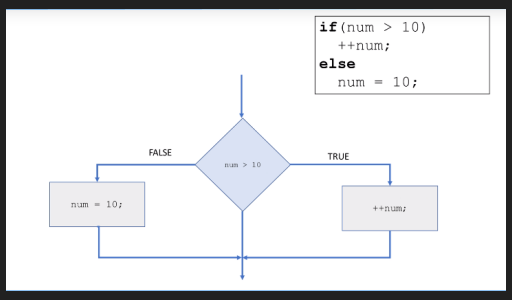
```
    const int min {10}, max {100};
    cout << "Enter a number between " << min << " and " << max << ": ";
    cin >> num;
    if (num >= min) {
        cout << "\n================if statment 1==================" << endl;
        cout << num << " is grater than " << min << endl;
        int diff {num - min};
        cout << num << " is " << diff << " greater than " << min << endl;
    }

    if (num <= max) {
        cout << "\n================if statment 2==================" << endl;
        cout << num << " is less than or equal to " << max << endl;
        int diff {max - num};
        cout << num << " is " << diff << " less than " << max << endl;
    }

    if (num >= min && num <= max) {
        cout << "\n================if statment 3==================" << endl;
        cout << num << " is in range " << endl;
        cout << "This means statement 1 and 2 must also display!" << endl;
    }
    if (num == min || num == max) {
        cout << "\n================if statment 4==================" << endl;
        cout << num << " is in boundaries " << endl;
        cout << "This means statement 1, 2 and 3 must also display!" << endl;
    }
    return 0;
}
/* OUTPUT:
Enter a number between 10 and 100: 100

================if statment 1==================
100 is grater than 10
100 is 90 greater than 10

================if statment 2==================
100 is less than or equal to 100
100 is 0 less than 100

================if statment 3==================
100 is in range
This means statement 1 and 2 must also display!

================if statment 4==================
100 is in boundaries
This means statement 1, 2 and 3 must also display!

*/
```

9.3 If else statement

- Syntax:

```
if (expr)
    statement;
else
    statement2;
```

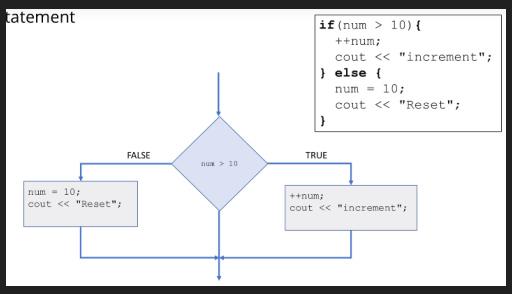
- If the expression is true then execute statement1.
- If the expression is false then execute statement2.
- Never forget the indentation.



9.3.1 Example of single if else statements

```
if (num > 10)
    cout << "num is greater than 10";
else
    cout << "num is NOT greater than 10";

if (health < 100 && heal_player)
    health = 100;
else
    ++health;
```

9.3.2 If else block



9.3.3 If else if construct

- Sometimes we need to check multiple conditions not just weather the expression is true but other characteristics, for this we use an if else if construct.

```
if (score > 90)
    cout << "A";
else if (score > 80)
    cout << "B";
```

```
else if (score > 70)
    cout << "C";
else if (score > 60)
    cout << "D";
else
    cout << "F";
cout << "Done";
```

9.3.4 Example

```
#include <iostream>
using namespace std;
int main() {
    int num{};
    const int target {10};
    cout << "Enter a number and I'll compare it to " << target << ": ";
    cin >> num;

    if (num >= target) {
        cout << "\n==========================" << endl;
        cout << num << " is greater than or equal to " << target << endl;
        int diff {num - target};
        cout << num << " is " << diff << " greater than " << target << endl;
    } else {
        cout << "\n==========================" << endl;
        cout << num << " is less than " << target << endl;
        int diff {target - num};
        cout << num << " is " << diff << " less than " << target << endl;
    }
    cout << endl;
    return 0;
}
/* OUTPUT:
Enter a number and I'll compare it to 10: -10

==========================
-10 is less than 10
-10 is 20 less than 10

*/
```

9.4 Nested if statement

- You can nest if statements within other if statements.

```
if (expr1)
    if (expr2)
        statement1;
    else
        statement2;
```

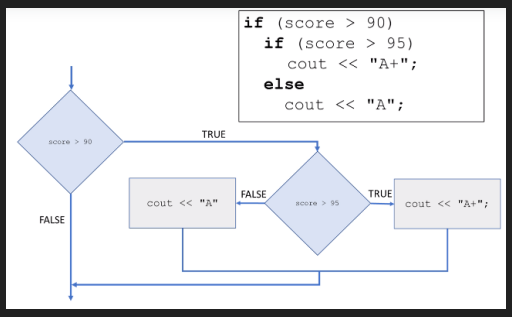
- If the expression is true then execute statement1.
- If the expression is false then execute statement2.

- Many times we need more logic to the if statement.
- In the above example the if statement expects one statement without the curly braces, however the if statement is a one compound statement and that is why the above code doesn't need curly braces.
- Notice, how does the computer know if the else statement belongs to the closest or farthest if? In C++ each else belogs to the closest if, so in the above example it belongs to the if holding the expr2.
- Remember to indent.

9.4.1 Example

- Example of scores:

```
if (score > 90)
    if (score > 95)
        cout << "A+";
    else
        cout << "A";
cout << "Sorry, No A";
```



- Example:

```
if (score_frank != score_bill) {
    if (score_frank > score_bill) {
        cout << "Frank wins" << endl;
    } else {
        cout << "Bill wins" << endl;
    }
} else {
    cout << "Looks like a tie!" << endl;
}
```

- Example of nested latter:

```
#include <iostream>
using namespace std;
int main() {
    int score {};
    cout << "Enter your score in the exam (0-100): ";
    cin >> score;
    char letter_grade {};

    if (score >= 0 && score <= 100) {
        if (score >= 90)
```

```
            letter_grade = 'A';
        else if (score >= 80)
            letter_grade = 'B';
        else if (score >= 70)
            letter_grade = 'C';
        else if (score >= 60)
            letter_grade = 'D';
        else
            letter_grade = 'F';
        cout << "Your grade is: " << letter_grade << endl;
        if (letter_grade == 'F')
            cout << "Bad grade." << endl;
        else
            cout << "Congrats." << endl;
    } else {
        cout << "Sorry, " << score << "is not in range." << endl;
    }
    return 0;
}
/* OUTPUT:
Enter your score in the exam (0-100): 90
Your grade is: A
Congrats.

*/
```

9.5 Switch-case statement

- There are some applications where you need to execute certain blocks of code depending on the value of a constant, for this you can use the `switch` statement.

```
switch (integer_control_expr) {
    case expression_1: statement_1; break;
    case expression_2: statement_2; break;
    case expression_3: statement_3; break;
    case expression_4: statement_4; break;
    ...;
    default: statement_default; break;
}
```

- Syntax: The switch keyword is followed by the control expression in parentheses. This control expression must evaluate to an integer type or an enumeration type. Then you have a series of case statements, the value of the control expression will be compared to the values contained in each case, these also need to be constants or literals pertaining to an enumeration or integral type. If no values match the switch will execute the default, this is like the else. The break statement in C++ are optional but best practice is to include them unless you have a very good reason not to. A break statement is not needed in the default case.

- If one case is followed by another such as: the switch will act as a logical or.

```
case '3':
case '4': cout << "3 or 4 selected";
```

It is the same as saying:

```
if ('3' || '4')
    cout << "3 or 4 selected";
```

- Once the switch has hit a match, no further cases will be checked.

9.5.1 Example

- Suppose `selection` is an character initialized to the character 2.

```
switch (selection) {
    case '1': cout << "1 selected"; // compare selection to 1, not true.
            break;
    case '2': cout << "2 selected"; // compare selection to 2, is true, execute the code from the c
            break;
    case '3': cout << "3 selected";
            break;
    case '4': cout << "4 selected";
            break;
    default: cout << "1,2,3,4 NOT selected";
}
```

- Example of fall-through:

```
#include <iostream>
using namespace std;
int main() {
    char selection {'2'};
    switch (selection) {
        case '1': cout << "1 selected\n";
                break;
        case '2': cout << "2 selected\n"; // this case evaluates 2 or 3 or 4 or 5.
        case '3': cout << "3 selected\n";
        case '4': cout << "4 selected\n";
        case '5': cout << "5 selected\n";
                break;
        case '6': cout << "4 selected\n";
                break;
        default: cout << "1,2,3,4 NOT selected\n";
    }
    return 0;
}
/* OUTPUT:
2 selected
3 selected
4 selected
5 selected

*/
```

- The switch statement is commonly used with enumeration, meaning the `enum`, for example:

```
#include <iostream>
using namespace std;
int main() {
    enum Color {
        red, green, blue
```

```
    };
    Color screen_color {green};
    switch (screen_color) {
        case red: cout << "red"; break;
        case green: cout << "green"; break;
        case blue: cout << "blue"; break;
        default: cout << "should never execute";
    }
    return 0;
}
/* OUTPUT:
green
*/
```

9.5.2 Review

- The control expression must evaluate to an integer type.
- The case expressions must be constant expressions that evaluate to integer or integers literals. They must be constants or literals, no variables.
- Once a match occurs all the code in the case statement is executed until a break is reached, then the switch terminates.
- Best practice is to provide break statement for each case.
- Best practice is also to provide `default` case, this is optional but should be handled in case it does get executed.

9.5.3 Example

- Switch with characters:

```
#include <iostream>
using namespace std;
int main() {
    char letter_grade{};
    cout << "Enter the letter grade you expect on the exam: ";
    cin >> letter_grade;
    switch (letter_grade) {
    case 'a':
    case 'A':
        cout << "90 or above." << endl;
        break;
    case 'b':
    case 'B':
        cout << "80-89" << endl;
        break;
    case 'c':
    case 'C':
        cout << "70-79" << endl;
        break;
    case 'd':
    case 'D':
        cout << "60-69" << endl;
        break;
```

```
        case 'f':
        case 'F': {
            char confirm{};
            cout << "Are you sure (Y/N)?";
            cin >> confirm;
            if (confirm == 'y' || confirm == 'Y') {
                cout << "0-60" << endl;
            } else {
                cout << "Illegal choice." << endl;
            }
            break;
        }
        default:
            cout << "Invalid input." << endl;
            break;
        }
        return 0;
    }
    /* OUTPUT:
    Enter the letter grade you expect on the exam: f
    Are you sure (Y/N)?y
    0-60

    */
```

- Switch with enumeration (`enum`):

```
    #include <iostream>
    using namespace std;
    int main() {
        enum Direction {
            left, right, up, down
        };
        Direction heading {left};

        switch (heading) {
            // some compilers won't let you compile if you don't handle every case. So it would not co
            case left:
                cout << "Going left" << endl;
                break;
            case right:
                cout << "Going right" << endl;
                break;
            case up:
                cout << "Going up" << endl;
                break;
            case down:
                cout << "Going down" << endl;
                break;
            default:
                cout << "OK" << endl;
        }
        return 0;
    }
    /* OUTPUT:
```

```
Going left
*/
```

9.6 Conditional operator

- Syntax is: `(cond_expr) ? expr1 : expr2;`
- This is frequently called the ternary operator.
- `cond_expr` evaluates to a boolean expression.
  - If `cond_expr` is true the value of `expr1` is returned.
  - If `cond_expr` is false the value of `expr2` is returned.
- Similar to an if-else construct in a single expression.
- Ternary operator.
- Very useful when used inline.
- Very easy to abuse! Best practice is to never nest a conditional operator within another one, this causes the code to become unreadable and difficult to manage.

9.6.1 Example

- Example:

```
#include <iostream>
using namespace std;
int main() {
    int a {10}, b {20};
    int score {92};
    int result {};
    result = (a > b) ? a : b; // false.
    cout << result << endl;
    result = (a < b) ? (a - b) : (a-b); // true.
    cout << result << endl;
    result = (b != 0) ? (a / b) : 0; // false.
    cout << result << endl;
    cout << ((score > 90)? "Excelent":"Good"); // true.
    return 0;
}
/* OUTPUT:
20
-10
0
Excelent
*/
```

- You can use conditional operators to write ifs in one single line:

```
#include <iostream>
using namespace std;
int main() {
    int num {};
    cout << "Enter an int: ";
    cin >> num;
```

```
    // usually written with ifs:
    if (num % 2 == 0)
        cout << num << " is even." << endl;
    else
        cout << num << " is odd." << endl,
    cout << endl;

    // you could do it with the conditional operator:
    cout << num << " is" << ((num % 2 == 0) ? " even." : " odd.") << endl;

    return 0;
}
/* OUTPUT:
Enter an int: 6
6 is even.
6 is even.

*/
```

- Another example:

```
#include <iostream>
using namespace std;
int main() {
    int num1 {}, num2 {};
    cout << "Enter two integers separated by a space: ";
    cin >> num1 >> num2;
    if (num1 != num2) {
        cout << "Largest: " << ((num1 > num2)?num1:num2) << endl;
    } else {
        cout << "The numbers are the same." << endl;
    }
    return 0;
}
/* OUTPUT:
Enter two integers separated by a space: 10 11
Largest: 11

*/
```

9.7 Looping

- Looping is also called iteration and is the third basic building block of programming.
  - Sequence, selection, iteration. Remember with these building blocks you can implement any algorithm.
- Iteration or repetition.
- Allows the execution of a statement or block of statements repeatedly.
- Loops are made up of a loop condition and the body which contains the statements to repeat.

9.7.1 Use cases of looping

Execute a loop:

- A specific number of times, such as the user enters the number of times.
- For each element in a collection of elements.
- While specific condition(s) remains true.
- Until a specific condition becomes false.
- Until we reach the end of some input stream.
- Forever (infinite loops) (for example operating systems).
- Many, many more use cases.

9.7.2 C++ looping constructs

- `for` loop:
  - Iterate specific number of times.
- Range-based for loop:
  - One iteration for each element in a range or collection.
  - Used with arrays and vectors for example.
- `while` loop:
  - Iterate while condition remains true.
  - Stop when the condition becomes false.
  - Check the condition at the beginning of every iteration.
- `do while` loop:
  - Iterate while a condition remains true.
  - Stop when the condition is false.
  - Check the condition at the end of every iteration.
  - The same as the while loop only difference is that the do while checks the condition after each iteration, the while loop checks conditions before each iteration.

9.8 `for` loop

- You can omit the curly braces if only one statement will be executed, if two or more are executed that will need to be enclosed in a block of code.

```
for (initalization; condition; increment)
    statement;
for (initalization; condition; increment) {
    statement1;
    statement2;
    statementn;
}
```

- The initialization is executed exactly once. Then the condition is checked, if its true it will execute the block of code and then the increment will take place, when the condition evaluates to false the loop terminates.

9.8.1 Examples

- Count from one to five:

```
    #include <iostream>
using namespace std;
int main() {
    int i {0};
    for (i = 1; i <= 5; ++i) {
        cout << i << endl;
    }

    return 0;
}
/* OUTPUT:
1
2
3
4
5
*/
```

- You can declare the conter variable (if you must use one) and initialize all inside the for loop, the advantage to this is that you don't need to declare it outside (like the previous example), however that variable will remain only within the score of the for loop, you cannot use it else where.

```
#include <iostream>
using namespace std;
int main() {
    for (int i {1}; i <= 5; ++i) {
        cout << i << endl;
    }
    for (int i = 1; i <= 5; ++i) {
        cout << i << endl;
    }
    cout << i << endl; // compiler error, i was not declared in this scope.
    return 0;
}
```

Correction:

```
#include <iostream>
using namespace std;
int main() {
    for (int i {1}; i <= 5; ++i) {
        cout << i << endl;
    }
    for (int i = 1; i <= 5; ++i) {
        cout << i << endl;
    }
    return 0;
}
/* OUTPUT:
1
2
3
```

```
4
5
1
2
3
4
5
*/
```

- Display even numbers from one to ten:

```
#include <iostream>
using namespace std;
int main() {
    for (int i {1}; i <= 10; ++i) {
        if (i % 2 == 0) {
            cout << i << endl;
        }
    }
    return 0;
}
/* OUTPUT:
2
4
6
8
10
*/
```

- Use for loops to iterate an array:

```
#include <iostream>
using namespace std;
int main() {
    int scores[] {100,90,87};
    for (int i {0}; i < 3; ++i) {
        cout << scores[i] << endl;
    }
    return 0;
}
/* OUTPUT:
100
90
87
*/
```

  - Very careful with array out of bounds errors.

- You can initialize many variables using the comma operator:

```
#include <iostream>
using namespace std;
int main() {
    for (int i {1}, j {5}; i <= 5; ++i, ++j) {
        cout << i << " * " << j << " : " << (i * j) << endl;
    }
```

```
    return 0;
}
/* OUTPUT:
1 * 5 : 5
2 * 6 : 12
3 * 7 : 21
4 * 8 : 32
5 * 9 : 45
*/
```

- Note that the associativity is right to left, and the result of the comma operator is the left most expression.
- In this case we just use them to initialize two variables and then to increment those two variables.

9.8.2 Other details

- The basic for loop is very clear and concise.
- Since the for loop's expressions are all optional, it is possible to have:
  - No initialization, No condition, and No increment.
  - This creates an infinite loop, and as we will see later on is the same as using the while loop with the condition being just true always.
  - You can also use doubles and other types as counters.
  - Best practice is to never include any expressions too complicated to document, if the expressions are too complicated consider using another looping construct.

```
for (;;) {
    cout << "Endless loop" << endl;
}
```

9.8.3 Examples

- Count from one to ten:

```
#include <iostream>
using namespace std;
int main() {
    for (int i {1}; i <= 10; ++i) {
        cout << i << endl;
    }
    return 0;
}
/* OUTPUT:
1
2
3
4
5
6
7
8
9
10

*/
```

- Count from one to ten every two:

```cpp
#include <iostream>
using namespace std;
int main() {
    for (int i {1}; i <= 10; i += 2) {
        cout << i << endl;
    }
    return 0;
}
/* OUTPUT:
1
3
5
7
9

*/
```

- For decrementing:

```cpp
#include <iostream>
using namespace std;
int main() {
    for (int i {10}; i > 0; --i) {
        cout << i << endl;
    }
    return 0;
}
/* OUTPUT:
10
9
8
7
6
5
4
3
2
1

*/
```

- Iterating a vector:

```cpp
#include <iostream>
#include <vector>
using namespace std;
int main() {
    vector<int> nums {10,20,30,40,50};
    for (unsigned i{0}; i < nums.size(); ++i) {
        cout << nums[i] << endl;
    }
    return 0;
}
/* OUTPUT:
```

```
10
20
30
40
50

*/
```

9.9 Range based for loop

- The idea with the range based for loop is to loop through a collection of elements and be able to easily access each element, without having to worry about the length of the collection or incrementing or decrementing looping variables or subscripting indexes.
- The syntax is:

```
for (var_type var_name : collection_name)
    statement1; // can use var_name.
    for (var_type var_name : collection_name) {
        statements; // can use var_name.
    }
```

- `var_type` will be bound to each element of the collection so it should be of the same type the collection of elements.
- Now when we access each variable name in the loop it will have a specific element in the collection.
- For example:

```
#include <iostream>
using namespace std;
int main() {
    int scores[] {100,90,97};
    for (int score : scores) {
        cout << score << endl;
    }
    return 0;
}
/* OUTPUT:
100
90
97

*/
```

- Instead of providing the `var_type` you can use the `auto` keyword, this allows the C++ compiler to deduce the type itself.

```
#include <iostream>
using namespace std;
int main() {
    int scores[] {100,90,97};
    for (auto score : scores) {
        cout << score << endl;
    }
```

```
    return 0;
}
/* OUTPUT:
100
90
97

*/
```

9.9.1 Examples

- Vector example:

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
    vector<double> temps {87.2, 77.1, 80.0, 72.5};
    double average_temp {};
    double running_sum {};
    for (auto temp : temps) {
        running_sum += temp;
    }
    average_temp = running_sum / temps.size();
    cout << average_temp << endl;
    return 0;
}
/* OUTPUT:
79.2
*/
```

- Intializer list:

```
#include <iostream>
using namespace std;
int main() {
    double average_temp {};
    double running_sum {};
    int size {};
    for (auto temp : {60.2, 80.1, 90.0, 78.2}) {
        running_sum += temp; size++;
    }
    average_temp = running_sum / size;
    cout << average_temp << endl;
    return 0;
}
/* OUTPUT:
77.125
*/
```

- Looping a string:

```
#include <iostream>
using namespace std;
int main() {
    for (auto c: "C++") {
```

```
        cout << c << endl;
    }
    return 0;
}
/* OUTPUT:
C
+
+

*/
```

9.10 While

- The while loop is an example of a pre-test loop because the expression is evaluated before executing the code.
- If the expression evaluates to true the code is executed, else it is not executed.
- Syntax is:

```
while (expression)
    statement;
while (expression) {
    statement (s);
}
```

- The `expression` has to evaluate to a boolean value. Followed by the statement or the block of statements.

9.10.1 Example

- Counting from 1-5.

```
#include <iostream>
using namespace std;
int main() {
    int i {1};
    while (i <= 5) {
        cout << i << endl;
        ++i; // increment
    }
    return 0;
}
/* OUTPUT:
1
2
3
4
5

*/
```

- Even numbers between 1 and 10.

```cpp
#include <iostream>
using namespace std;
int main() {
    int i {1};
    while (i <= 10) {
        if (i % 2 == 0)
            cout << i << endl;
        ++i;
    }
    return 0;
}
/* OUTPUT:
2
4
6
8
10

*/
```

- Array example:

```cpp
#include <iostream>
using namespace std;
int main() {
    int scores[] {100,90,87};
    int i {0};
    while (i < 3) {
        cout << scores[i] << endl;
        ++i;
    }
    return 0;
}
/* OUTPUT:
100
90
87

*/
```

- Input validation:

```cpp
#include <iostream>
using namespace std;
int main() {
    int number {};
    cout << "Enter an integer less than 100: ";
    cin >> number;
    while (number < 100) {
        cout << "Enter an integer less than 100: ";
        cin >> number;
    }
    cout << "Thanks" << endl;
    return 0;
}
/* OUTPUT:
```

```
Enter an integer less than 100: 100
Enter an integer less than 100: 99
Thanks

*/
```

9.11 do-while loop

- Syntax is:

```
do {
    statements;
} while (expression);
```

- The do-while loop is a post-test loop, because the expression is tested at the end of each iteration.
- This means that the loop body is garanteed to be executed at least once.

9.11.1 Examples

- Input validation:
  - In the below code, notice that variable declarations cannot be done inside the do-while code body, if you do this you will get a compiler error.
  - You cannot declare variables that are involved in the expression inside the code body.

```
#include <iostream>
using namespace std;
int main() {
    int number {};
    do {
        cout << "Enter an integer between 1 and 5: ";
        cin >> number;
    } while (number <= 1 || number >= 5);
    cout << "Thanks" << endl;
    return 0;
}
/* OUTPUT:
Enter an integer between 1 and 5: 0
Enter an integer between 1 and 5: 9
Enter an integer between 1 and 5: 3
Thanks

*/
```

- Input validation:
  - Remember to declare the variable used in the expression outside the do-while loop. You can declare any variable you want inside the du-while loop as long as you don't use it in the expression.

```
#include <iostream>
using namespace std;
int main() {
    char selection {};
```

```
    do {
        double width {}, heigth {};
        cout << "Enter width and heigth: ";
        cin >> width >> heigth;
        double area {width * heigth};
        cout << "The area is " << area << endl;
        cout << "Calculate another? (Y/N): ";
        cin >> selection;
    } while (selection == 'Y' || selection == 'y');
    cout << "Thanks" << endl;
    return 0;
}
/* OUTPUT:
Enter width and heigth: 1 2
The area is 2
Calculate another? (Y/N): y
Enter width and heigth: 2 3
The area is 6
Calculate another? (Y/N): n
Thanks
*/
```

9.12 `continue` and `break`

- The continue and break statements can be used within all C++ loop constructs, they add more explicit control over the looping behaviour.
- `continue`:
  - When a continue statement is executed inside a loop, no further statements in the body of the loop are executed.
  - Control immediately goes directly to the beggining of the loop for the next iteration.
  - "Skip this iteration and continue with the next one."
- `break`
  - When a break statement is executed in a loop no further statements in the body of the loop are executed.
  - Loop is immedietly terminated.
  - Control immediately goes to the statement following the loop construct.
- Don't overuse the break and continue statement.

9.12.1 Example

- We have data and if a piece of data is -1 we don't want to process it. We want to stop if we find -99 in the vector.

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
    vector<int> values {1,2,-1,-99,7,8,10};
    for (auto val : values) {
```

```
        if (val == -99) {
            break;
        } else if (val == -1) {
            continue;
        } else {
            cout << val << endl;
        }
    }
    return 0;
}
/* OUTPUT:
1
2

*/
```

9.13 Infinite loops

- An infinite loop is a loop whose condition expression always evaluates to true.
- Usually this is unintended and a programmer error.
- Sometimes programmers use infinite loops and include and break statements in the body to control them.
- Sometimes infinite loops are exactly what we need, such as in:
  - Even loop in an event-driven program: common in embeded systems or mobile devices where the program listens and reacts to movement detected by the mouse or other such sensors, and this continues as long as the program is running.
  - Operating system: an infinite loop executes and breaks until you shut down your computer.

9.13.1 Example

- Infinite for loop:

```
for (;;) {
    do_something;
}
```

- Infinite while loop:

```
while (true) {
    do_something;
}
// or
while (10 < 12) { // 10 is always les than 12.
    do_something;
}
```

- Infinite do-while loop:

```
do {
    do_something;
} while (true);
```

- Breaking the infinite while loop:

```cpp
#include <iostream>
using namespace std;
int main() {
    while (true) {
        char again {};
        cout << "Do you want to loop again? (Y/N): ";
        cin >> again;
        if (again == 'N' || again == 'n') {
            break;
        }
    }
    return 0;
}
/* OUTPUT:
Do you want to loop again? (Y/N): y
Do you want to loop again? (Y/N): y
Do you want to loop again? (Y/N): y
Do you want to loop again? (Y/N): n

*/
```

9.14 Nested loops

- Loops can be nested one inside another.
- We can nest loops as deep as we need.
- Nested loops can be as many levels deep as the program needs.
- Very useful with multi-dimensional data structures. Such as 2D arrays, 2D vectors, etcetera.
- Outer loop vs. Inner loop.

9.14.1 Examples

- 2D Looping:

```cpp
#include <iostream>
using namespace std;
int main() {

    for (int outer_val {1}; outer_val <= 2; ++outer_val) {
    for (int inner_val {1}; inner_val <= 3; ++inner_val) {
                cout << outer_val << ", " << inner_val << endl;
            }
        }
    return 0;
}
/* OUTPUT:
1, 1
1, 2
1, 3
2, 1
2, 2
2, 3
```

```
*/
```

- Multiplication table:

```
#include <iostream>
using namespace std;
int main() {
    for (int num1 {1}; num1 <= 10; ++num1) {
        for (int num2 {1}; num2 <= 10; ++num2) {
            cout << num1 << " * " << num2 << " = " << num1 * num2 << endl;
        }
    }
    return 0;
}
/* OUTPUT:
1 * 1 = 1
1 * 2 = 2
1 * 3 = 3
1 * 4 = 4
1 * 5 = 5
1 * 6 = 6
1 * 7 = 7
1 * 8 = 8
1 * 9 = 9
1 * 10 = 10
2 * 1 = 2
2 * 2 = 4
2 * 3 = 6
2 * 4 = 8
2 * 5 = 10
2 * 6 = 12
2 * 7 = 14
2 * 8 = 16
2 * 9 = 18
2 * 10 = 20
3 * 1 = 3
3 * 2 = 6
3 * 3 = 9
3 * 4 = 12
3 * 5 = 15
3 * 6 = 18
3 * 7 = 21
3 * 8 = 24
3 * 9 = 27
3 * 10 = 30
4 * 1 = 4
4 * 2 = 8
4 * 3 = 12
4 * 4 = 16
4 * 5 = 20
4 * 6 = 24
4 * 7 = 28
4 * 8 = 32
4 * 9 = 36
```

4 * 10 = 40
5 * 1 = 5
5 * 2 = 10
5 * 3 = 15
5 * 4 = 20
5 * 5 = 25
5 * 6 = 30
5 * 7 = 35
5 * 8 = 40
5 * 9 = 45
5 * 10 = 50
6 * 1 = 6
6 * 2 = 12
6 * 3 = 18
6 * 4 = 24
6 * 5 = 30
6 * 6 = 36
6 * 7 = 42
6 * 8 = 48
6 * 9 = 54
6 * 10 = 60
7 * 1 = 7
7 * 2 = 14
7 * 3 = 21
7 * 4 = 28
7 * 5 = 35
7 * 6 = 42
7 * 7 = 49
7 * 8 = 56
7 * 9 = 63
7 * 10 = 70
8 * 1 = 8
8 * 2 = 16
8 * 3 = 24
8 * 4 = 32
8 * 5 = 40
8 * 6 = 48
8 * 7 = 56
8 * 8 = 64
8 * 9 = 72
8 * 10 = 80
9 * 1 = 9
9 * 2 = 18
9 * 3 = 27
9 * 4 = 36
9 * 5 = 45
9 * 6 = 54
9 * 7 = 63
9 * 8 = 72
9 * 9 = 81
9 * 10 = 90
10 * 1 = 10
10 * 2 = 20
10 * 3 = 30

```
10 * 4 = 40
10 * 5 = 50
10 * 6 = 60
10 * 7 = 70
10 * 8 = 80
10 * 9 = 90
10 * 10 = 100

*/
```

- 2D arrays - set all elements to 1000:

```
#include <iostream>
using namespace std;
int main() {
    int grid[5][3] {};
    // initialize everything.
    for (int row {0}; row < 5; ++row) {
        for (int col {0}; col < 3; ++col) {
            grid[row][col] = 1'000;
        }
    }
    // display elements.
    for (int row {0}; row < 5; ++row) {
        for (int col {0}; col < 3; ++col) {
            cout << grid[row][col] << ", ";
        }
        cout << endl;
    }
    return 0;
}
/* OUTPUT:
1000, 1000, 1000,
1000, 1000, 1000,
1000, 1000, 1000,
1000, 1000, 1000,
1000, 1000, 1000,

*/
```

- 2D vector - display elements:

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
    vector<vector<int>> vector_2d {
        {1,2,3},
        {10,20,30,40},
        {100,200,300,400,500}
    };
    for (auto vec: vector_2d) {
        for (auto val: vec) {
            cout << val << " ";
        }
        cout << endl;
```

```
    }
    return 0;
}
/* OUTPUT:
1 2 3
10 20 30 40
100 200 300 400 500

*/
```

Chapter 10

Characters and strings

10.1 Character functions

- The cctype library includes very useful functions that allow us to work with characters, such as the following:
  - Functions for testing characters.
  - Functions for converting character case.
- In order to use those functions you must include the library.

```
#include <cctype>
```

- The testing functions accept a single character and all evaluate to a boolean value, the conversion functions return the converted character.
- Some of the testing functions are:
  - To use them you must pass the character to evaluate.

`isalpha(c)`	True if c is a letter.
`isalnum(c)`	True if c is a letter or digit.
`isdigit(c)`	True if c is a digit.
`islower(c)`	True if c is lowercase letter.
`isprint(c)`	True if c is a printable character.
`ispunct(c)`	True if c is a punctuation character.
`isupper(c)`	True if c is an uppercase letter.
`isspace(c)`	True if c is whitespace.

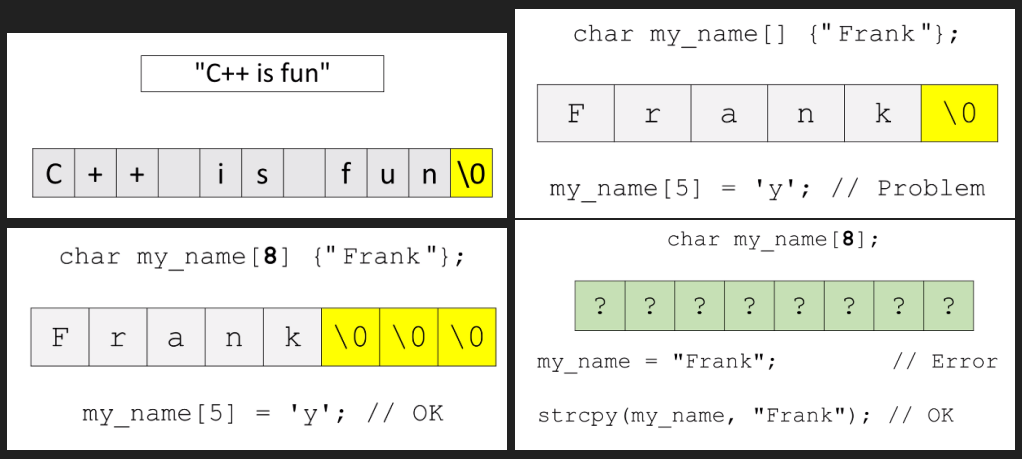
- Character conversion functions:

`tolower(c)`	returns lowercase of c, if the function can't convert, it will return to the character that was passed in.
`toupper(c)`	returns uppercase of c, if the function can't convert, it will return to the character that was passed in

10.2 C-Style strings

- A C-Style string is a sequence of characters:
  - Stored contiguous in memory.
  - Implemented as an array of characters. You can access each character using the array subscript syntax.

  - Terminated by a null character (null).
    - Null character with a value of zero.
  - Refered to as zero or null terminated strings.
- String literal:
  - Sequence of characters in double quotes.
  - Constant.
  - Terminated with a null character.
- How C++ stores strings in memory:



- The cstring library contains functions that work with C-style strings, these serve for: (include the library `#include <cstring>`)
  - Copying.
  - Concatenation.
  - Comparison.
  - Searching.
  - And others.
- C++ has a library called cstdlib, containing functions that convert C-Style strings to other types, such as:
  - Integer.
  - Float.
  - Long.
  - Etc.

10.2.1 Examples

- Example of string functions:

```
#include <iostream>
#include <cstring>
using namespace std;
int main() {
    char str[80];
```

```
    strcpy(str, "Hello "); // copy.
    strcat(str, "there "); // concatenate.
    cout << strlen(str); // 10
    strcmp(str, "Another"); // -1
    return 0;
}
/* OUTPUT:
12
*/
```

10.3 Working with C-Style strings examples

- Example:

```
#include <iostream>
#include <cstring>
using namespace std;
int main() {
    char first_name[20] {};
    char last_name[20] {};
    char full_name[20] {};
    char temp[50] {};
    cout << "Please enter your first and last name: ";
    cin >> first_name >> last_name;
    cout << "--------------------------" << endl;
    cout << "Hello, " << first_name << " your first name has " << strlen(first_name) << " character

    strcpy(full_name, first_name);
    strcat(full_name, " ");
    strcat(full_name, last_name);

    cout << full_name << endl;
    return 0;
}
/* OUTPUT:
Please enter your first and last name: David Corzo
--------------------------
Hello, David your first name has 5 characters.
David Corzo

*/
```

- In the last example we read in the firs name and the last name separately, the program delimited the space entered by the user. In this example we will read in the full name with the space.

```
#include <iostream>
#include <cstring>
using namespace std;
int main() {
    char full_name[50] {};
    cout << "Please enter your first and last name: ";
    cin.getline(full_name, 50); // it will stop at 50 chars or when it finds the return character.
    cout << full_name << endl;
    return 0;
```

```
}
/* OUTPUT:
Please enter your first and last name: David Corzo
David Corzo

*/
```

- Comparing strings:

```
#include <iostream>
#include <cstring>
using namespace std;
int main() {
    char str1[] {"Hello world\n"};
    char str2[] {"Hello world\n"};
    if (strcmp(str1, str2) == 0) {
        cout << "The strings are the same." << endl;
    }
    return 0;
}
/* OUTPUT:
The strings are the same.

*/
```

- Iterating and changing strings:

```
#include <iostream>
#include <cstring>
using namespace std;
int main() {
    char str1[] {"Hello world."};
    for (size_t i {0}; i < strlen(str1); ++i) {
        if (isalpha(str1[i])) {
            str1[i] = toupper(str1[i]);
        }
    }
    cout << str1 << endl;
    return 0;
}
/* OUTPUT:
HELLO WORLD.

*/
```

10.4 C++ strings

- `std::string` is a class in the C++ Standard template library.
  - You must include the library `#include <string>`.
  - You must include the `std` namespace.
  - Stored contiguous in memory.
  - Dynamic size.
  - Work with input and output streams.

  - There are lots of useful member functions.
  - Our familiar operators can be used (`+`, `=`, `<`, `<=`, `>`, `>=`, `+=`, `==`, `!=`, `[]`, ...). They also work with the insertion and extraction operator. This is another example of operator overloading.
  - Can be converted to C-Style strings if needed.
  - Safer to use.

- Declaring and initializing:
  - You can initialize a C++ string using any initialization style.
  - C++ strings are always initialized, unlike C-style strings that were never initialized.

```
#include <iostream>
#include <string>
using namespace std;
int main() {
    string s1; // Empty.
    string s2 {"Frank"}; // Frank
    string s3 {s2}; // Frank
    string s4 {"Frank", 3}; // Fra
    string s5 {s3,0,2}; // Fr
    string s6 (3, 'X'); // XXX
    return 0;
}
```

- With C++ strings you can use the assignment operator.

```
#include <iostream>
#include <string>
using namespace std;
int main() {
    string s1;
    s1 = "C++";
    string s2;
    s2 = s1;
    return 0;
}
```

- Concatenation can be done with the `+` operator.
  - In the below example if we would have tried to do something as:

    ```
    sentence = "C++" + " is powerful";
    ```

  - We would have had a compiler error, because as far as C++ is concerned these are two C-Style literals, not `std::string` objects, thus addition is not concatenation.
  - A combination of C-style strings and C++ strings is ok, the code below does this, however you cannot add two C-Style strings.

```
#include <iostream>
using namespace std;
int main() {
    string part1 {"C++"};
    string part2 {"is powerful"};
    string sentence;
    sentence = part1 + " " + part2 + " language";
    cout << sentence << endl;
```

```
    return 0;
}
/* OUTPUT:
C++ is powerful language
*/
```

- Accessing characters [] and at() method:
  - The difference between the [] and the .at() method is that the .at() method in the `std::string` performs bounds checking and the [] does not. As a best practice, always use the .at() method.

```
#include <iostream>
using namespace std;
int main() {
    string s1;
    string s2 {"Frank"};
    cout << s2[0] << endl; // F
    cout << s2.at(0) << endl; // F
    return 0;
}
/* OUTPUT:
F
F

*/

#include <iostream>
#include <string>
using namespace std;
int main() {
    string s1 {"Frank"};
    for (char c : s1) {
        cout << c << endl;
    }
    return 0;
}
/* OUTPUT:
F
r
a
n
k

*/
```

- Comparing `std::string` objects can be done with the `==`, `!=`, `>`, `>=`, `<`, `<=` operators. We can perform comparisons in the following cases:
  - When we have two `std::string` objects.
  - When we have one `std::string` object and C-style string literal.
  - When we have `std::string` object and C-style string variable.
  - We cannot perform this comparison if both are string literals.
  - Greater than, less than, or equal are done by comparing the values in the ASCII table.

```cpp
#include <iostream>
#include <string>
using namespace std;
int main() {
    cout << boolalpha;
    string s1 {"Apple"};
    string s2 {"Banana"};
    string s3 {"Kiwi"};
    string s4 {"apple"};
    string s5 {s1};
    cout << (s1 == s5) << endl; // true
    cout << (s1 == s2) << endl; // false
    cout << (s1 != s2) << endl; // true
    cout << (s2 > s1) << endl; // true
    cout << (s4 < s5) << endl; // false
    cout << (s1 == "Apple") << endl; // true
    return 0;
}
/* OUTPUT:
true
false
true
true
false
true

*/
```

- You can extract substrings from a string using the `substr()` method.
  - This method extracts a substring from a `std::string`.

    ```
    object.substr(start_index, lenght);
    ```

```cpp
#include <iostream>
#include <string>
using namespace std;
int main() {
    string s1 {"This is a test"};
    cout << s1.substr(0,4) << endl; // This
    cout << s1.substr(5,2) << endl; // is
    cout << s1.substr(10,4) << endl; // test
    return 0;
}
/* OUTPUT:
This
is
test

*/
```

- To find a substring in a `std::string` you can use the `find()` method.
  - This method returns the index of a substring in a `std::string`.

    ```
    object.find(search_string);
    ```

```cpp
#include <iostream>
#include <string>
using namespace std;
int main() {
    string s1 {"This is a test"};
    cout << s1.find("This") << endl;
    cout << s1.find("This") << endl;
    cout << s1.find("is") << endl;
    cout << s1.find("test") << endl;
    cout << s1.find('e') << endl;
    cout << s1.find("is",4) << endl; // The 4 represents from what index forward you want to start
    cout << s1.find("XX") << endl; // its not found so the value returned is string::npos
    return 0;
}
/* OUTPUT:
0
0
2
10
11
5
18446744073709551615

*/
```

  - There is also an `.rfind` method that starts searching from the end of the string.

- You can also remove characters using the `erase()` and `clear()` methods.
  - This methods removes a substring of characters from a `std::string`.

    ```cpp
    object.erase(start_index, lenght);
    ```

```cpp
#include <iostream>
#include <string>
using namespace std;
int main() {
    string s1 {"This is a test"};
    cout << s1.erase(0,5) << endl;
    cout << s1.erase(5,4) << endl;
    s1.clear(); // clears the string.
    return 0;
}
/* OUTPUT:
is a test
is a

*/
```

- To obtain the length of the string use:

  ```cpp
  object.length();
  ```

- To cocatenate you can use the `+=` operator:

  ```cpp
  s1 += " James"; // concatenates s1 and " James".
  ```

- Input with C++ strings:

- The `cin` reads until it finds a space, sometimes we want to read up until the user enters the return character. For this we can use `getline(cin, s1);`.

```cpp
#include <iostream>
#include <string>
using namespace std;
int main() {
    string s1;

    getline(cin, s1); // reads entire line.
    cout << s1 << endl;

    getline(cin, s1, 'x'); // you can add a delimeter.
    cout << s1 << endl;
    return 0;
}
/* OUTPUT:
Hello world
Hello world
Hello world x
Hello world

*/
```

Chapter 11

Functions

11.1 What is a function?

- In C++ programs we typically make use of:
  - C++ standard library (functions and classes).
  - Third-party library (functions and classes).
  - Our own functions and classes.
- Functions allow the modularization of a program.
  - Which allow us to separate code into logical self-contained units.
  - These units can be reused.
- Why we want to use functions?
  - Which code is better?
  - Without functions:

```
int main() {
    // read input
    statement1;
    statement2;
    statement3;
    statement4;
    // process input
    statement5;
    statement6;
    statement7;
    // provide input
    statement8;
    statement9;
    statement10;
    return 0;
}
```

  - With functions:

```
int main() {
    // read input
    read_input();
    // process input
```

```
    process_input();
    // provide input
    provide_input();
    return 0;
}
```

  - Functions allow for abstraction.
  - Functions allow us to use the same code again and again and not having to be copying and pasting everything.

- What is a function? The boss/worker analogy.
  - Write your code to the function specification.
  - Understand what the function does.
  - Understand what information the function needs.
  - Understand what the function returns.
  - Understand any errors the function my produce.
  - Understand any performance constraints.
  - Don't worry about how the function works internally:
    * Unless you are the one writing the function.
    * This concept is called information hidding.

- Example: `<cmath>`
  - Common mathematical calculations.
  - Global functions called as:
    * Global functions mean they are available to the entire program.

```
function_name(argument);
function_name(argument1, argument2, ...);
```

- User defined functions:
  - Implement only the functions you need, use the functions already implemented if posible.
  - `https://en.cppreference.com/w/cpp/header` You can check functions.

11.1.1 Example

- Math:

```
#include <iostream>
#include <cmath>
using namespace std;
int main() {
    double num {12.5};
    cout << sqrt(num) << endl;
    return 0;
}
/* OUTPUT:
3.53553
*/
```

11.2 Function definition

Functions follow four main parts:

- Functions perform operations.
- Once a function is defined you can call it from anywhere in the program.
- Functions need to be declared before they are used.
- Name:
  - The name of the function.
  - Function names must be descriptive.
  - Same rules as for variable names.
  - Should be meaningful.
  - Usually a verb or verb phrase.
- Parameter list:
  - The variables passed into the function.
  - You can pass no variables.
  - Their types must be specified.
- Return type:
  - The type of the data that is returned from the function.
  - It's possible that the function does not return anything, in that case the type is `void`.
- Body:
  - The statements that are executed when the function is called.
  - In curly braces {}.

11.2.1 Example

- Example of function:

```
int function_name(int a) {
    return a;
}
```

- Example of area of the circle:

```
#include <iostream>
using namespace std;

const double pi {3.4159};

double calc_area_circle(double radious) {
    return pi * radious * radious;
}

void area_circle() {
    double radious {};
    cout << "Enter the radious of the circle: ";
```

```
    cin >> radious;

    cout << "The area of a circle with radious " << radious << " is " << calc_area_circle(radious)
}

double calc_volume_cylinder(double radious, double height) {
    return calc_area_circle(radious) * height;
}

void volume_cylinder() {
    double radious {};
    double height {};
    cout << "Enter the radious of the cylinder: ";
    cin >> radious;
    cout << "Enter the height of the cylinder: ";
    cin >> height;
    cout << "The volume of a cylinder with radious " << radious << " and height " << height << " is
}

int main() {
    area_circle();
    volume_cylinder();
    return 0;
}
/* OUTPUT:
Enter the radious of the circle: 78
The area of a circle with radious 78 is 20782.3
Enter the radious of the cylinder: 78
Enter the height of the cylinder: 90
The volume of a cylinder with radious 78 and height 90 is 1.87041e+06

*/
```

11.3 Function prototypes

- The compiler must know about a function before it is used.
- This implies that either you define functions before calling them or you can use *function prototypes.*
- Defining functions before they are called is OK for small programs, however not at all practical for larger programs.
- Using function prototypes allows us to define the functions anywhere in the code, because they are technically being 'declared' beforehand.
  - Function prototypes tells the compiler what it needs to know without a full function definition.
  - Also called "forward declarations" since you are telling the compiler how the function is going to look like and the compiler will enforce those rules.
  - Function prototypes are placed at the beginning of a program.
  - Also used in out own header files (.h). Functions defined in header files must contain their corresponding function prototypes in the same file.
- Syntax of function prototype:

```
int function_name(); // prototype.

int function_name() { // function definition.
    statement(s);
    return 0;
}
```

- Function prototypes can include parameter names.

```
int function_name(int); // prototype with out names.
int function_name(int a); // prototype with parameter names.
int function_name(int a) {
    statement(s);
    return 0;
}
```

- With `void` the return type is optional:

```
void function_name(); // prototype.
void function_name() {
    statement(s);
    return; // optional
}
```

- Function parameters help the compiler enforce the rules, for example what parameters it takes in, what types are those parameters, etcetera.

```
#include <iostream>
using namespace std;

void say_hello();

int main() {
    say_hello(); // OK
    say_hello(100); // error
    cout << say_hello(); // error, cout expects to print something and there is no return value
    return 0;
}
/* OUTPUT:

*/
```

11.4 Function parameters and the return statement

11.4.1 Function parameters

- Function parameters:
  - When we call a function we can pass in data to that function.
  - In the function call they are called arguments.
  - In the function definition they are called parameters.

  - They must match in number, order, and type.

```
#include <iostream>
using namespace std;

int add_numbers(int, int); // prototype.

int main() {
    int result {0};
    result = add_numbers(100,200); // call.
    cout << result << endl;
    return 0;
}

int add_numbers(int a, int b) { // function definition.
    return a + b;
}

/* OUTPUT:
300
*/
```

- If the compiler knows how to convert a type it will convert.

```
#include <iostream>
using namespace std;

void say_hello(string);

void say_hello(string name) {
    cout << "Hello " << name << endl;
}

int main() {
    say_hello("C++"); // Sending a string literal not a std::string. The compiler will convert.
    return 0;
}
/* OUTPUT:
Hello C++

*/
```

- Pass-by-value:
  - When you pass data into a function it is passeded-by-value.
  - A copy of the data is passed to the function.
  - Whatever changes you make to the parameter in the function does NOT affect the argument that was passed in.
  - This is good because the original data is not touched by the function if something goes wrong, and bad because sometimes copying the function parameters is expensive in terms of memory space, time and money. And sometimes you really do need to change the data that is passed in.
  - Formal vs. Actual parameters:
    * Formal parameters: the parameters defined in the function header.
    * Actual parameters (arguments): the parameters used in the function call.

- In C++ the actual parameters are passed by value or copied to the formal parameters.

```cpp
#include <iostream>
using namespace std;

void param_test(int formal) { // formal is a copy if actual.
    cout << formal << endl; // 50
    formal = 100; // only changes the local (formal) copy.
    cout << formal << endl; // 100.
}

int main() {
    int actual {50};
    cout << actual << endl; // 50.
    param_test(actual); // pass in 50 to param_test.
    cout << actual << endl; // actual (50) did not change.
    return 0;
}
/* OUTPUT:
50
50
100
50

*/
```

11.4.2 Function return statement

- If a function returns a value of a specific type then it must use a `return` statement that returns a value.
- If a function does not return a value (`void`) then the `return` statement is optional.
- `return` statement can occur anywhere in the body of the function, but it is usually used in the end of the function.
- `return` statement immediately exits the function.
- We can have multiple `return` statements in a function:
  - Avoid many return statements in a function.
- The return value is the result of the function call.

11.5 Default arguments

- When a function is called, all arguments must be supplied.
- Sometimes some of the arguments have the same values most of the time.
- We can tell the compiler to use default values if the arguments are not supplied.
- Default values can be in the prototype or definition, not both:
  - Best practice is to have them in the prototype.
  - Must appear at the tail end of the parameter list.
- Can have multiple default values.
  - Must appear consecutively at the tail end of the parameter list.

11.5.1 Example

- Example:

```
#include <iostream>
using namespace std;

double calc_cost(double base_cost, double tax_rate = 0.06);

double calc_cost(double base_cost, double tax_rate) {
    return base_cost += (base_cost * tax_rate);
}

int main() {
    double cost {0};
    cost = calc_cost(200.0); // will use the default tax.
    cout << cost << endl;
    cost = calc_cost(100.0, 0.08); // will use 0.08 not the default
    cout << cost << endl;
    return 0;
}
/* OUTPUT:
212
108

*/
```

11.6 Overloading functions

- Sometimes we want to use a function to perform operations in several data types, such as the print function would need to admit any type of parameter.
- We can have functions that have different parameter lists that have the same name.
- This is a great example of abstraction, since we just think of "print" and not worry about the data type being passed in.
- This is a type of polymorphism.
  - We can have the same name work with different data types to execute similar behavior.
- The compiler must be able to tell the functions apart based on the parameter list and argument supplied.
- All overloaded functions must be implemented separately, this could be solved with templates.
- If the function don't have any parameters and just vary on the return type a compiler error will be produced.
- The ease of use of overloading functions is just the abstraction, you can just think of a concept and not have to worry in which data type you implemented the function in, the compiler will deduce that for you.
- Be careful when using default values in more than one overloaded function, the compiler will not know which one to use, and this can cause a compiler error.
- Consider the following:

```cpp
#include <iostream>
using namespace std;

int add_numbers(int a, int b);
double add_numbers(double a, double b);

double add_numbers(double a, double b) {
    return a + b;
}
int add_numbers(int a, int b) {
    return a + b;
}

int main() {
    cout << add_numbers(10,20) << endl;
    cout << add_numbers(10.1,67.9) << endl;
    return 0;
}
/* OUTPUT:
30
78

*/
```

11.6.1 Example

```cpp
#include <iostream>
#include <vector>
using namespace std;

void print(int);
void print(double);
void print(string);
void print(string, string);
void print(vector<string>);

void print(int num) {
    cout << "int:" << num << endl;
}
void print(double num) {
    cout << "double: " << num << endl;
}
void print(string str) {
    cout << "string: " << str << endl;
}
void print(string str1, string str2) {
    cout << "str1: " << str1 << ", str2: " << str2 << endl;
}
void print(vector<string> vec) {
    for (auto v : vec) {
        cout << v << endl;
    }
}
```

```
int main() {
    print(100); // int -> print(int)
    print('A'); // char -> print(int) char is promoted to int.
    print(123.5); // double -> print(double)
    print(123.3F); // float -> print(double) float is promoted to double.
    print("C-style string"); // C-style string is converted to a C++ string.
    vector<string> vec {"ABC", "DEF", "GHI"};
    print(vec);
    return 0;
}
/* OUTPUT:
int:100
int:65
double: 123.5
double: 123.3
string: C-style string
ABC
DEF
GHI

*/
```

11.7 Passing arrays to functions

- We can pass an array to a function by providing square brackets in the formal parameter description.

```
void print_array(int numbers[]);
```

- The array elements are NOT copied, the function does not know how many times to iterate through the array because what is passed in is an address to the array not a copy of the array.
- Since the array name evaluates to the location of the array in memory, this address is what is copied.
- So the function has no idea how many elements are in the array since all it knows is the location of the first element (the name of the array).
- So when we pass arrays to functions we must also pass in the size of the array in order to know how much time to iterate.
- Example:

```
#include <iostream>
using namespace std;

void print_array(int numbers[], size_t size);

int main() {
    int my_numbers[] {1,2,3,4,5};
    print_array(my_numbers, 5);
    return 0;
}

void print_array(int numbers[], size_t size) {
    for (size_t i{0}; i < size; ++i) {
```

```
        cout << numbers[i] << endl;
    }
}
/* OUTPUT:
1
2
3
4
5

*/
```

- In this case since the array is not copied but instead C++ works with the original array, we sometimes don't want to risk changing or altering the array passed in. Since we are passing the location of the array the function can modify the actual array.
  - For this we can tell the compiler that function parameters are `const` (read-only).
  - This could me useful in the `print_array` function since if should NOT modify the array.

```
void print_array(const int numbers[], size_t size) {
    for (size_t i{0}; i < size; ++i) {
        cout << numbers[i] << endl;
    }
}
```

11.8 Pass by reference

- Sometimes we want to be able to change the actual parameter from within the function body.
- In order to achieve this we need the location or address of the actual parameter.
- We say how this is the effect with array, but what about other variable types?
- We can use reference parameters to tell the compiler to pass in a reference to the actual parameter.
- The formal parameter will now be an alias for the actual parameter.
- Syntax is: `return_type func_name(type &var_name);`
- Example:

```
#include <iostream>
using namespace std;

void scale_number(int &num);

int main() {
    int number {1000};
    scale_number(number);
    cout << number << endl;
    return 0;
}

void scale_number(int &num) {
    if (num > 100) {
        num = 100;
```

```
    }
}

/* OUTPUT:
100

*/
```

- Example:

```
#include <iostream>
using namespace std;

void swap(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
}

int main() {
    int x{10}, y{20};
    cout << x << " " << y << endl;
    swap(x,y);
    cout << x << " " << y << endl;
    return 0;
}
/* OUTPUT:
10 20
20 10

*/
```

- In the following we intend to pass a vector by value in to the function print which would mean that it will make a copy of all that data inside the function. We could better pass the location of the vector into the function and modify the original data:
  - The issue now would be that we could pass by reference and risk modifying the vector in an unintended way, for this we can make the data being passed in constant.
  - This is the best of both worlds because we are passing the reference (which is more efficient) and at the same time ensuring no changes are made to the vector.

```
#include <iostream>
#include <vector>
using namespace std;

void print(const vector<int> &v) {
    for (auto num : v) {
        cout << num << endl;
    }
}

int main() {
    vector<int> data {1,2,3,4};
    print(data);
    return 0;
```

```
}
/* OUTPUT:
1
2
3
4

*/
```

11.9 Scope rules

- C++ uses scope rules to determine where an identifier can be used.
- C++ uses static or lexical scoping.
- C++ has two main scopes: Local or block scope:
  - Local or block scopes:
    - Identifiers declared in a block {}.
    - Function parameters have block scope.
    - Only visible within the block {} where declared.
    - Function local variables are only active while the function is executing.
    - Local variables are not preserved between function calls.
    - With nested blocks inner blocks can 'see' but outer blocks cannot.
    - When a function is called you can think of the function as being activated and the function parameters are bound to storage. The parameters become alive and the their lifetime is the time the function executes, once the function completes the function is deactivated and these variables and parameters are no longer alive.
- Static local variables:
  - There is one kind of variable whose value is preserved between many function calls.

    ```
    static int value {10};
    ```
  - It is a variable whose lifetime is the lifetime of the program.
  - It is only visible to the statements in the function body.
  - These variables can be very helpful when you need some values to be preserved in a function without having to pass them in each time.
  - Static local variables are only initialized once, if no initialization is provided they are set to 0.
- Identifiers with global scope.
  - Identifiers declared outside any function or class.
  - Visible to all parts of the program after the global identifier has been declared.
  - Global constants are OK.
  - Best practice is not to use global variables.
- You can create a new scope with the {}.

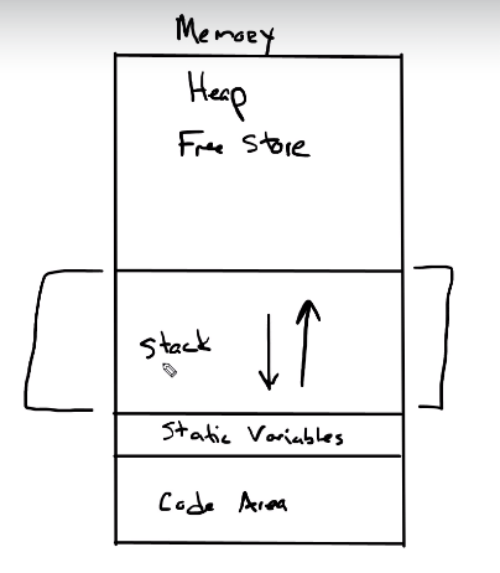
```
#include <iostream>
using namespace std;
int main() {
    int num1 {100}; // local to main.
    int num2 {500}; // local to main.
```

```
    cout << "num1: " << num1 << " {" << endl;
    {
        int num1 {200};
        cout << "\tnum1: " << num1 << endl;
        cout << "\tnum2: " << num2 << endl;
    }
    cout << '}' << endl;
    return 0;
}
/* OUTPUT:
num1: 100 {
        num1: 200
        num2: 500
}

*/
```

- One particular block can see outwardly while the out most blocks cannot see in to the most nested blocks.

11.10 How do function calls work?

- Functions use an area in memory called the function call stack:
  - Analogous to a stack of books.
  - Operations on a stack push and pop.
  - In a program a push element in to the call stack is called a stack frame.
- Stack frame or activation record:
  - Functions must return control to function that called it.
  - Each time a function is called we create an new activation record and push it on to the stack.
  - When a function terminates we pop the activation record and return.
  - Local variables and function parameters are located on the stack.
- The stack size if finite:
  - If you call too many functions you might run out of stack space producing a stack overflow error which will crash your program.
- Illustration:



11.11 Inline functions

- As we've seen, a function requires space in the call stack, the creation of a stack frame and the deletion of the stack frame once the function is done.
- Sometimes the function is so simple the process a function has to undergo to become a function in memory is ineficient and not worth the code.
- Function calls have certaing amount of overhead.
- You saw what happens on the call stack.
- Sometimes we have simple functions.
- We can suggest to the compiler to compile them 'inline':
  - Avoid function call overhead.
  - Generate inline assembly code.
  - Faster.
  - Could cause code bloat.
- Compilers optimizations are very sophisticated:
  - Will likely make simple functions inline even without your suggestion.
- Syntax is:

```
inline int add_numbers(int a, int b) {
    return a + b;
}
int main() {
    int result{0};
    result = add_numbers(100,200);
    return 0;
}
```

11.12 Recursive functions

- A recursive function is a function that calls itself.
  - Either directly or indirectly through another function.
- Recursive problem-solving:
  - Base case.
  - Divide the rest of the problem into subproblems and do a recursive call.
- There are many problems that lend themselves to recursive solutions.
- Mathematics: factorial, Fibbonacci, fractals.
- Searching and sorting: binary search, search trees, ...
- Example of Factorial:

$$n! = n \times (n - 1);$$

```cpp
#include <iostream>
using namespace std;
unsigned long long factorial(unsigned long long n) {
    if (n == 0) {
        return 1; // base case.
    }
    return n * factorial(n-1); // recursive call.
}
int main() {
    cout << factorial(8) << endl;
    return 0;
}
/* OUTPUT:
40320

*/
```

- Important notes:
  - If recursion doesn't eventually stop you will have infinite recursion.
  - Recursion can be resource intensive (take up too much memory).
  - Remember the base case(s):
    - It terminates the recursion.
  - Only use recursive solutions when it makes sense.
  - Anything that can be done recursively can be done iteratively.
    - Stack overflow error.

Chapter 12

Pointers and references

12.1 What is a pointer?

- A variable.
  - Whose value is an address.
- What can be at that address?
  - Another variable.
  - A function.
- Pointers have a memory location that is bound to, type and has a value, this value is address in memory.
- If x is an integer variable and its value is 10 then I can declare a pointer that points to it.
- To use the data that the pointer is pointing to you must know its type.

12.1.1 Why use pointers?

- Can't I just use the variable or the function itself?
  - Yes, but you can't always do that.
- Inside functions, pointers can be used to access data that are defined outside the function. Those values may not be in scope so you can't access them by their name.
- Pointers can be used to operate on arrays very efficiently.
- We can allocate memory dynamically on the heap or free store.
  - This memory doesn't even have a variable name.
  - The only way to get to it is via a pointer.
- With object-oriented programming, pointers are how polymorphism works.
- Can access specific addresses in memory.
  - Useful in embedded and system applications.

12.2 Declaring pointer variables

- We declare pointer variables in very much the same way any other variable would, except an asterisk must be between the type and the identifier:

```
variable_type *pointer_name;
```

```
int *int_ptr; // pointer to int.
double *double_ptr; // pointer to double.
char *char_ptr; // pointer char.
std::string *string_ptr; // pointer to a std::string.
```

  - Keep in mind that there are many conventions to declaration of pointers, such as the one where the asterisk is next to the type `int* name;`, others where the asterisk is previous to the name `int *name;` or even a space from each `int * name;`

- Intializing pointer variables to "point no where" or null:

  - If you don't initialize your pointers they will have garbage data.
  - In this case that data will be addresses.
  - Just as an int was as convention initialized to a value, say 0, a pointer is usually initialized to null which means it doesn't point to any memory location.
  - You do this with the `nullptr` or the `NULL` keywords.

```
variable_type *pointer_name {nullptr};
```

```
int *int_ptr {};
double *double_ptr {nullptr};
char *char_ptr {nullptr};
string *string_ptr {nullptr};
```

- Always initialize variable pointers to 'point no where 'or null.

  - Always initialize.
  - Uninitialized pointers contain garbage data and can 'point anywhere' in memory.
  - Initializing to zero or `nullptr` (C++11) represents address zero.
    - Implies that the pointer is 'pointing nowhere'.
  - If you don't initialize a pointer to point to a variable or function then you should initialize it to `nullptr` to 'make it null'.

12.3 Accessing the pointer address and storing address in a pointer

- The address operator.

  - Variables are stored in unique addresses.
  - Unary operator.
  - Evaluates to the address of its operand.
    - Operand cannot be a constant or expression that evaluates to temp values.

```cpp
#include <iostream>
using namespace std;
int main() {
    int num{10};
    cout << "Value of num is: " << num << endl;
    cout << "Size of num is: " << sizeof num << endl;
    cout << "Address of num is: " << &num << endl;
    return 0;
}
/* OUTPUT:
Value of num is: 10
Size of num is: 4
Address of num is: 0x61fe1c
*/
```

- Example:

```cpp
#include <iostream>
using namespace std;
int main() {
    int *p;
    cout << "Value of p is: " << p << endl; // garbage data.
    cout << "Address of p is: " << &p << endl; // value of p.
    cout << "Size of p is: " << sizeof p << endl; // size of p.
    p = nullptr;
    cout << "Value of p is: " << p << endl;
    return 0;
}
/* OUTPUT:
Value of p is: 0x10
Address of p is: 0x61fe18
Size of p is: 8
Value of p is: 0

*/
```

12.3.1 sizeof a pointer variable

- Don't confuse the size of a pointer variable and the size of what it points to.
- All pointers in a program have the same size.
- They may be pointing to a very large or very small types.

```cpp
int *p1 {nullptr};
double *p2 {nullptr};
unsigned long long *p3 {nullptr};
vector<string> *p4 {nullptr};
string *p5 {nullptr};
```

12.3.2 Storing an address in a pointer variable?

Typed pointers.

- The compiler will make sure that the address stored in a pointer variable is of the correct type.

```
int score {10};
double high_temp {100.7};
int *score_ptr {nullptr};
score_ptr = &score;
score_ptr = &high_temp; // compiler error because high_temp is a double and score_ptr is an int.
```

12.3.3 & the address of operator

- Pointers are variables so they can change.
- Pointers can be null.
- Pointers can be uninitialized.

12.4 Dereferencing the pointer

- Access the data we're pointing to is called dereferencing a pointer.
- If `score_ptr` is a pointer and has a valid address.
- Then you can access the data at the address contained in the `score_ptr` using the dereferencing operator.

```
#include <iostream>
using namespace std;
int main() {
    int score {100};
    int *score_ptr {&score};
    cout << *score_ptr << endl; // 100
    *score_ptr = 200;
    cout << *score_ptr << endl;
    cout << score << endl;
    return 0;
}
/* OUTPUT:
100
200
200
*/
```

  - Declaring and dereferencing is done using the asterisk, C++ has received some criticism about this, but once you understand where and how to use the asterisk you'll be fine.

- Example:

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
    vector<string> stooges {"Larry", "Moe", "Curly"};
    vector<string> *vector_ptr {nullptr};
    vector_ptr = &stooges;
    cout << "First stooge: " << (*vector_ptr).at(0) << endl;
    cout << "Stooges: ";
    for (auto stooge: *vector_ptr) {
        cout << stooge << " ";
```

```
    }
    cout << endl;
    return 0;
}
/* OUTPUT:
First stooge: Larry
Stooges: Larry Moe Curly
*/
```

12.5 Dynamic Memory Allocation

- Allocating storage from the heap at runtime.
- We often don't knot how much storage we need until we need it.
- We can allocate storage for a variable at run time.
- Recall C++ arrays:
  - We had to explicitly provide the size and it was fixed.
  - But vectors grow and shrink dynamically.
- We can use pointers to access newly allocated heap storage.

12.5.1 Allocating and deallocating memory

- The `new` keyword
  - Using the new keyword to allocate storage.

```
#include <iostream>
using namespace std;
int main() {
    int *int_ptr {nullptr};
    int_ptr = new int; // allocate an integer on the heap.
    cout << int_ptr << endl; // address of int_ptr.
    cout << *int_ptr << endl; // garbage. Notice the dereferencing.
    *int_ptr = 100;
    cout << *int_ptr << endl; // 100.
    return 0;
}
/* OUTPUT:
0x25824f0
39331936
100
*/
```

  - If you loose the pointer because it goes out of scope or other such incidents, that is called a memory leak and you lost the only way you have to access that memory.
  - You also must deallocate the pointer after you are done using it.
- The `delete` keyword
  - The delete keyword is used to deallocate allocated space.

```
int *int_ptr {nullptr}; // allocate an integer on the heap.
delete int_ptr; // frees the allocated storage.
```

- The `new[]` keyword
  - The `new[]` is used to allocate an array.

```
int *array_ptr {nullptr};
int size {};
cout << "How big do you want the array: ";
cin >> size;
array_ptr = new int[size]; // allocate array on the heap.
```

  - Keep in mind that these brackets must be empty.

- The `delete[]` keyword is used to deallocate storage of an array.

```
int *array_ptr {nullptr};
int size {};
cout << "How big do you want the array: ";
cin >> size;
array_ptr = new int[size]; // allocate array on the heap.
delete[] array_ptr; // free allocated storage.
```

  - Keep in mind that these brackets must be empty.

- When you allocate dynamically you are allocating into the heap or the free store, the stack houses the pointer to the dynamically allocated data.
  - Example:

```
#include <iostream>
using namespace std;
int main() {
    size_t size{0}; // allocated on the stack.
    double *temp_ptr {nullptr}; // allocated on the stack.
    cout << "How many temps: ";
    cin >> size;
    temp_ptr = new double[size]; // allocated on the heap.
    cout << temp_ptr << endl;
    delete[] temp_ptr; // dealocated on the heap.
    return 0;
}
```

12.6 Relationship between arrays and pointers

- The value of an array name is the address of the first element in the array.
- The value of a pointer variable is an address.
- If the pointer points to the same data types as the array element then the pointer and array name can be used interchangeably (almost).

```
#include <iostream>
using namespace std;
int main() {
    int scores[] {100,95,89};
    cout << scores << endl;
    cout << *scores << endl;
    int *score_ptr {scores};
    cout << score_ptr << endl;
```

```
    cout << *score_ptr << endl;
    return 0;
}
/* OUTPUT:
0x61fe0c
100
0x61fe0c
100
*/
```

- We can also use array subscripting on a pointer using the square brackets operator.

```
#include <iostream>
using namespace std;
int main() {
    int scores[] {100,95,89};
    int *score_ptr {scores};
    cout << score_ptr[0] << endl;
    cout << score_ptr[1] << endl;
    cout << score_ptr[2] << endl;
    return 0;
}
/* OUTPUT:
100
95
89
*/
```

- You can perform pointer arithmetic, which is adding numbers to a pointer:

```
#include <iostream>
using namespace std;
int main() {
    int scores[] {100,95,89};
    int *score_ptr {scores};
    cout << score_ptr << endl;
    cout << (score_ptr + 1) << endl;
    cout << (score_ptr + 2) << endl;
    return 0;
}
/* OUTPUT:
0x61fe0c
0x61fe10
0x61fe14
*/

#include <iostream>
using namespace std;
int main() {
    int scores[] {100,95,89};
    int *score_ptr {scores};
    cout << *score_ptr << endl;
    cout << *(score_ptr + 1) << endl;
    cout << *(score_ptr + 2) << endl;
    return 0;
}
```

```
/* OUTPUT:
100
95
89
*/
```

12.6.1 Subscript and offset notation equivalence

- You can write this in two ways:

```
int array_name[] {1,2,3,4,5};
int *pointer_name {array_name};
```

- Subscript notation:

```
array_name[index];
pointer_name[index];
```

- Offset notation:

```
*(array_name + index);
*(pointer_name + index);
```

12.7 Pointer arithmetic

- Pointers can be used in:
  - Assignment expressions.
  - Arithmetic expressions.
  - Comparison expressions.
- C++ allows pointer arithmetic.
- Pointer arithmetic only makes sense with raw arrays.

12.7.1 ++ and –

- ++ increments a pointer to point to the next array element.

```
int_ptr++;
```

- – decrements a pointer to point to the previous array element.

```
int_ptr--;
```

- Keep in mind that in pointer arithmetic, adding one means adding whatever number of bytes the elements occupy in order to get to the next element.

12.7.2 + and -

- + increment pointer by some number: (`n * sizeof(type)`)

```
int_ptr += n; or int_ptr = int_ptr + n;
```

- - decrement pointer by some number: (`n * sizeof(type)`)

```
int_ptr -= n; or int_ptr = int_ptr - n;
```

12.7.3 Subtracting two pointers

- Determine the number of elements between the pointers.
- both pointers must point to the same data type:

```
int n = int_ptr2 - int_ptr1;
```

12.7.4 Compare two pointers == and !=

- Determine if two pointers point to the same location.
  - Does not compare the data where they point.

```
#include <iostream>
using namespace std;
int main() {
    string s1 {"Frank"};
    string s2 {"Frank"};
    string *p1 {&s1};
    string *p2 {&s2};
    string *p3 {&s1};
    cout << boolalpha;
    cout << (p1 == p2) << endl;
    cout << (p1 == p3) << endl;
    return 0;
}
/* OUTPUT:
false
true
*/
```

12.7.5 Comparing the data pointers point to

- Determine if two pointers point to the same data.
- You must compare the referenced pointers.

```
#include <iostream>
using namespace std;
int main() {
    string s1 {"Frank"};
    string s2 {"Frank"};
    string *p1 {&s1};
    string *p2 {&s2};
    string *p3 {&s1};
    cout << boolalpha;
    cout << (*p1 == *p2) << endl;
    cout << (*p1 == *p3) << endl;
    return 0;
}
/* OUTPUT:
false
true
*/
```

12.7.6 Examples

```
#include <iostream>
using namespace std;
int main() {
    int scores[] {100,95,89,68,-1};
    int *score_ptr {scores};
    while (*score_ptr != -1) {
        cout << *score_ptr << endl;
        score_ptr++;
    }
    return 0;
}
/* OUTPUT:
100
95
89
68
*/
```

```
#include <iostream>
using namespace std;
int main() {
    char name[] {"Frank"};
    char *char_ptr1 = &name[0];
    char *char_ptr2 = &name[3];
    cout << "In the string " << name << ", " << *char_ptr2 << " is " << (char_ptr2 - char_ptr1) << " cha
    return 0;
}
/* OUTPUT:
In the string Frank, n is 3 characters away from F
*/
```

12.8 Passing pointers to a function

const and pointers:

- There are several ways to qualify pointers using const.
  - Pointers to constants.
  - Constant pointers.
  - Constant pointers to constants.
- Pointers to constants:
  - The data pointed to by the pointers is constant and cannot be changed.
  - The pointer itself can change and point somewhere else.
  - Pointers to constants follow the syntax: `const type *var_name`.

```
int high_score {100};
int low_score {65};
const int *score_ptr {&high_score};
*score_ptr = 86; // error, changing the data being pointed to.
score_ptr = &low_score; // Ok.
```

- Constant pointers:
  - The data pointed to by the pointers can be changed.
  - The pointer itself cannot change and point somewhere else.
  - Const pointers follow the syntax: `type const *var_name`.

```
int high_score {100};
int low_score {65};
int const *score_ptr {&high_score};
*score_ptr = 86; // Ok.
score_ptr = &low_score; // Error, changing the pointer.
```

12.9 Passing pointers to functions

- Pass-by-reference with pointer parameters.
- We can use pointers and the dereference operator to achieve pass-by-reference.
- The function parameter is a pointer.
- The actual parameter can be a pointer of address of a variable.
- Example:

```
#include <iostream>
using namespace std;

void double_data(int *int_ptr);

void double_data(int *int_ptr) {
    *int_ptr *= 2;
}

int main() {
    int value {10};
    cout << value << endl; // 10
    double_data(&value);
    cout << value << endl; // 20
    return 0;
}
/* OUTPUT:
10
20
*/
```

- Example swap two variables:

```
#include <iostream>
using namespace std;

void swap(int *a, int *b) {
    int temp {*a};
    *a = *b;
    *b = temp;
}
```

```
int main() {
    int x {100}, y {200};
    cout << "x: " << x << endl;
    cout << "y: " << y << endl;
    swap(&x, &y);
    cout << "x: " << x << endl;
    cout << "y: " << y << endl;
    return 0;
}
/* OUTPUT:
x: 100
y: 200
x: 200
y: 100
*/
```

- Example of passing pointer to vector:

```
#include <iostream>
#include <vector>
using namespace std;

void display(const vector<string> const *v) { // the pointer is constant and the data is constant.
    for (auto str : *v) {
        cout << str << ' ';
    }
    cout << endl;
}

int main() {
    vector<string> stooges {"Larry", "Moe", "Curly"};
    display(&stooges);
    return 0;
}
/* OUTPUT:
Larry Moe Curly
*/
```

- Example or updating pointers:

```
#include <iostream>
using namespace std;

void display(int *array, int sentinel) {
    while (*array != sentinel) {
        cout << *array++ << endl; // dereference the pointer and print it, then increment the point
    }
    cout << endl;
}

int main() {
    int scores[] {100,98,97,79,85,-1};
    display(scores, -1);
    return 0;
}
/* OUTPUT:
```

```
*/
```

12.10 Returning a pointer from a function

- In C++ functions can return pointers.

```
type *function();
```

- Should return pointers to:
  - Memory dynamically allocated in the function.
  - To data that was passed in.
- Never return a pointer to a local function variable.
- Example, return the pointer to the largest int:

```
#include <iostream>
using namespace std;

int *largest_int(int *int_ptr1, int *int_ptr2) {
    if (*int_ptr1 > *int_ptr2) {
        return int_ptr1;
    } else {
        return int_ptr2;
    }
}

int main() {
    int a {100}, b {200};
    int *largest_ptr {nullptr};
    largest_ptr = largest_int(&a, &b);
    cout << *largest_ptr << endl; // 200
    return 0;
}
/* OUTPUT:
200
*/
```

- Example, returning dynamically allocated memory:

```
#include <iostream>
using namespace std;

int *create_array(size_t size, int init_value = 0) {
    int *new_storage {nullptr};
    new_storage = new int[size];
    for (size_t i {0}; i < size; ++i ) {
        *(new_storage + i) = init_value;
    }
    return new_storage;
}
```

```
int main() {
    int *my_array;
    my_array = create_array(100,200);
    delete[] my_array;
    return 0;
}
/* OUTPUT:
*/
```

- Never return a pointer to a local variable:

```
int *dont_do_this() {
    int size {};
    return &size;
}

int *or_this() {
    int size {};
    int *int_ptr {&size};
    return int_ptr;
}
```

  - Notice this will compile just fine, since the address returned is the address of an integer, however, all variables are deleted when they go out of scope and you return a pointer to a deleted memory location.

12.11 Potential pointer pitfalls

- Uninitialized pointers.
  - Contain garbage, the pointer might be pointing to a very important area in memory and cause serious crashes even in the OS.
  - Sometimes makes debugging much harder because sometimes your program does not crash and when you add somethin it unexpectedly crashes and you think it is because of the new change but its actually a bug that has been in the code for a long time.

```
int *int_ptr; // pointing anywhere.
int *int_ptr = 100; // Hopefully a crash.
```

- Dangling pointers.
  - Pointers that are pointing to memory that is no longer valid or released memory:
    - For example: 2 pointers point to the same data.
    - 1 pointer releases the data with delete.
    - The other pointer accesses the release data.
    - This causes undefined behaviour.
  - Pointer that points to memory that is invalid:
    - We saw this when we returned a pointer to a function's local variable.
- Not checing if new failed to allocate memory.
  - If `new` fails an exception is thrown.
  - We can use exception handling to catch exceptions.
  - Dereferencing a null pointer will cause your program to crash.

  - This is good in testing but bad in production.
- Leaking memory.
  - Forgetting to release dynamically allocated memory with delete.
  - If you lose your pointer to the storage allocated on the heap you have not way to get to that storage again.
  - The memory is orphaned or leaked.
  - One of the most common pointer problems.

12.12 What is a reference?

- An alias for a variable, not a pointer.
- Must be initialized to a variable when declared.
- Cannot be null.
- Once initialized cannot be made to refer to a different variable.
- Very useful as function parameters.
- Might be helpful to think of a reference as a constant pointer that is automatically dereferenced.

12.12.1 Using references in range-based for loops

- In the next example, we iterate a vector but are unable to change the data.

```
#include <iostream>
#include <vector>
using std::vector;
using std::string;
using std::cout;
using std::endl;

int main() {
    vector<string> stooges {"Larry", "Moe", "Curly"};
    for (auto str: stooges) {
        str = "Funny"; // changes the copy.
    }
    for (auto str: stooges) {
        cout << str << endl; // Larry, Curly, Moe
    }
}

/* OUTPUT:
Larry
Moe
Curly
*/
```

- Using references we can change the data.

```
#include <iostream>
#include <vector>
using std::vector;
```

```
using std::string;
using std::cout;
using std::endl;

int main() {
    vector<string> stooges {"Larry", "Moe", "Curly"};
    for (auto &str: stooges) {
        str = "Funny"; // changes the actual.
    }
    for (auto str: stooges) {
        cout << str << endl; // Funny, Funny, Funny
    }
}
/* OUTPUT:
Funny
Funny
Funny
*/
```

- Be careful using the `const` qualifier in range based for loops if you plan to change the data.

```
vector<string> stooges {"Larry", "Moe", "Curly"};
for (auto const &str: stooges) {
    str = "Funny"; // compiler error.
}
```

- It makes sense to use const if you don't want to change the data, using references you do not copy the elements, just the elements.

```
vector<string> stooges {"Larry", "Moe", "Curly"};
for (auto const &str: stooges) {
    cout << str << endl; // Larry, Moe, Curly
}
```

12.12.2 Examples

- Example: references are aliases to a variable.

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
    int num {100};
    int &ref {num};
    cout << num << endl;
    cout << ref << endl;

    num = 200;
    cout << num << endl;
    cout << ref << endl;

    ref = 300;
    cout << num << endl;
    cout << ref << endl;
}
/* OUTPUT:
```

```
100
100
200
200
300
300
*/
```

- References in range based for loops:

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
    vector<string> stooges {"Larry", "Moe", "Curly"};
    for (auto str: stooges) {
        str = "Funny"; // changing a copy.
    }
    for (auto str: stooges) {
        cout << str << " ";
    }
    cout << endl;
    for (auto &str: stooges) {
        str = "Funny"; // changing the actual.
    }
    for (auto str: stooges) {
        cout << str << " ";
    }
    cout << endl;
    return 0;
}
/* OUTPUT:
Larry Moe Curly
Funny Funny Funny
*/
```

12.13 L-values and R-values

12.13.1 L-values

- Values that have names and are addressable.
- Modifiable if they are not constants.

```
int x {100}; // x is an l-value.
x = 1000;
x = 1000 + 20;
string name; //name is an l-value.
name = "Frank";
```

- l-value:
  - Values that have names and are addressable.
  - Modifiable if they are not constants.

```
100 = x; // 100 is not an r-value.
(1000+20) = x; // (1000 + 20) is an l-value.
```

- r-values:
  - A value that is not an l-value. We can define it like this using exclusion.
  - On the right hand side of an assignment expression.
  - A literal.
  - A temporary which is intended to be non-modifiable.

```
string name;
name = "Frank";
"Frank" = name; // "Frank" is an r-value.
int max_num = max(20,30); // max(20,30) is an r-value.
```

  - r-values can be assigned to l-values explicitly.

```
int x {100};
int y {0};
y = 100; // r-value 100 assigned to l-value y.
x = x + y; // r-value (x-y) assigned to l-value x.
```

- References from the perpective of l and r values:
  - The references we've used are l-value references.
  - Because we are referencing l-values.

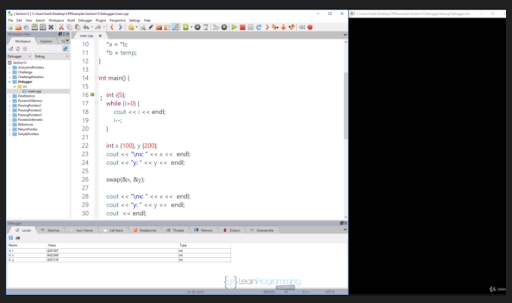
```
int x {100};
int &ref1 = x; // ref1 is reference to l-value.
ref1 = 1000;
int &ref2 = 100; // error: 100 is an r-value.
```

  - The same is true when we pass-by-reference to a function:

```
int square(int &n) {
    return n*n;
}
int num {10};
square(num); // ok.
square(5); // error: can't reference r-value 5.
```

12.14 Using the debugger

- Using F5 you will compile and run your program, your program will stop when it encounters a break point.
- On the break point you can see the local variables that exist, the functions as well as the locations of these elements.



- Step in: if the line is a function it allows us execute a break point inside the function.
- Step out: if you've pressed step in, steping out will continue where you left off.
- Continue/Next: will go to the next line.
- Watches: alloc you to check for conditions.

12.15 Section recap

- When to use pointers vs. reference parameters:
  - Pass by value.
    - When the function does not modify the actual parameter, and the parameter is small and efficient to copy like simple types (int, char, double, etc.)
  - Pass by reference using a pointer:
    - When the function does moddify the actual parameter, and the parameter is expensive to copy.
    - Its OK for a pointer to be null, references cannot be null.
  - Pass by reference usign a pointer to const:
    - Pass by reference using a pointer to `const`
      - When the function does not modify the actual parameter, and the parameter is expensive to copy, and its ok for the pointer to be a null value.
  - Pass by reference using a const pointer to const:
    - When the fucntion does not modify the actual parameter, adn the parameter is expensive to copy, and its OK for the pointer to be a null value, you don't want to modify the pointer itself.
  - Pass by reference using a reference:
    - When the function does modify the actual parameter, and the parameter is expensive to copy, and the parameter will never be a null value.
  - Pass by reference using a const reference:
    - When the function does not modify the actual parameter, and the parameter is expensive to copy, and the parameter will never be a null value.