**C++ Notes**

**Last Modified: 2023 September 8**

**Chapter 1: Theory**

**This section includes some theory questions, such as algorithms and understanding of the basic theory. But, some practical understandings (i.e., what is the use of comments?) are discussed in Chapter 3.**

* What is C++?
  + C++ is a cross-platform language that can be used to create high-performance applications.
  + C++ gives programmers a high level of control over system resources and memory.
  + The language was updated 4 major times in 2011, 2014, 2017, and 2020 to C++11, C++14, C++17, C++20.
* Why use C++?
  + C++ is an **object-oriented programming language** which gives a clear structure to programs and allows code to be reused, lowering development costs.
* Difference between C and C++
  + C++ was developed as an extension of C, and both languages have almost the same syntax.
  + The main difference between C and C++ is that C++ support classes and objects, while C does not.
* Why use visual code as an editor or an IDE (Integrated Development Environment)?
  + It is easy to install, and it is cross-platform. It runs well on Windows, Mac, and Linux.
* What is a compiler?
  + A compiler is a software that compiles our codes into code that can directly run on the hardware or binary executable format.
  + Testing our code on multiple compilers can make our code more portable.
  + Example: Mingw (GCC), Msvc, Clang llvm
* What is compile error and runtime error?
  + **Compile Error:** Occurs when you violate the rules of writing syntax, such as missing a semicolon, a parenthesis or using a reserved word. The compiler detects these errors and prevents the code from running.
  + **Runtime Error:** Occurs during the execution of the code, such as dividing by zero, null pointer or logic error. These errors are usually not detected by the compiler but cause the program to produce wrong results or terminate abnormally.
* How is a cpp file executed?
  + When we compile our program, the compiler converts our code into binary format, which is unreadable by humans but readable by machines. It is converted into binary format because it is easily understandable by the CPU.

A computer screen with a computer code

Description automatically generated

* When we run our program, the binary file is loaded up in memory (RAM), which is the program area in the following graph. CPU starts running the program statements by statements. It starts at the top (in our example, allocating space to store our integer ***a***). If a variable value is not specified, some junk value will be stored instead (e.g., ***c***). The CPU keeps a return address for a function to know where it should return after executing the function.

A computer screen shot of a computer

Description automatically generated

* What is Statement?
  + A statement is a basic unit of computation in a C++ program. Every C++ program is a collection of statements organized in a certain way to achieve some goal.
  + Statements end with a semicolon in C++ (***;***).
  + Statements are executed in order from top to bottom when the program is run.
  + Execution keeps going until there is a statement causing the program to terminate or run another sequence of statements.
* What is Variable?
  + A named piece of memory that you use to store specific types of data.
  + All C++ variables must be identified with unique names.These unique names are called **identifiers**.
  + Identifiers can be short names (like x and y) or more descriptive names (age, sum, totalVolume).
* What are Core Features?
  + Core features are a basic building block of the C++ programming language. For example, How to define variables, How to use a function in C++, etc.
* What is Standard Library?
  + A set of highly specialized components that we can easily used in C++ programs.
  + E.g., ***#include <iostream>***, ***#include <string>***
* What is STL (Standard Template Library)?
  + A part of C++ standard library, but it is collection of container and algorithm types. It is used to implement common data structures and algorithms in C++ programs.
  + Advantage: It provides a way to write generic, reusable code that can be applied to different data types.
  + E.g., ***List***, ***Vector***, ***Map***, ***Sort***, ***Find***, ***Transform***
* What is an Escape Sequence?
  + Escape sequence forces the cursor to change its position to the beginning of the next line on the screen.

|  |  |
| --- | --- |
| **Escape Sequence** | **Description** |
| **\t** | Creates a horizontal tab (4 spaces). |
| **\n** | Inserts a new line. |
| **\\** | Inserts a backslash character. (\) |
| **\'** | Inserts a single quote character. (') |
| **\"** | Inserts a double quote character. (**")** |

* How do we represent data in memory?
  + IEEE 754 is a method used to convert all data into 0 and 1 to represent it in the memory.
* What is ASCII code?
  + ASCII was among the first encodings to represent text in a computer.
  + It falls short when it comes to representing languages other than English (e.g., Chinese)
  + There are better ways to represent text that is meant to be seen in different languages, one of the most common being **Unicode**.

**Chapter 2: Configuration Setting**

* How to download C++ compilers
  + Download Mingw and Clang llvm in one place: <https://winlibs.com/>
* How to test whether the compiler GCC has been installed and added to the path (Windows)
  + *g++ --version* & *clang++ --version* in cmd

A screenshot of a computer program

Description automatically generated

**2.1 Visual Studio**

* How to check whether Visual Studio has been appropriately downloaded in Windows
  + Open Developer Command Prompt for VS and type cl.exe

A screenshot of a computer program

Description automatically generated

**2.2 Visual Studio Code**

* How to **run a cpp file** in Visual Studio Code?

1. Install C/C++ Extension.
2. Terminal -> Configure Tasks -> Choose the preferred compiler.

A screenshot of a computer

Description automatically generated

1. A JSON file will be created, which is our configuration to configure which compilers VS code uses to compiler our *main.cpp* file.
2. To run the cpp file in a particular C++ version: google GCC C++ <version>, e.g., GCC C++ 20
3. From <https://stackoverflow.com/questions/66975491/how-to-use-c-20-in-g>, we know we should add *-std=c++20* in JSON file.

A screen shot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

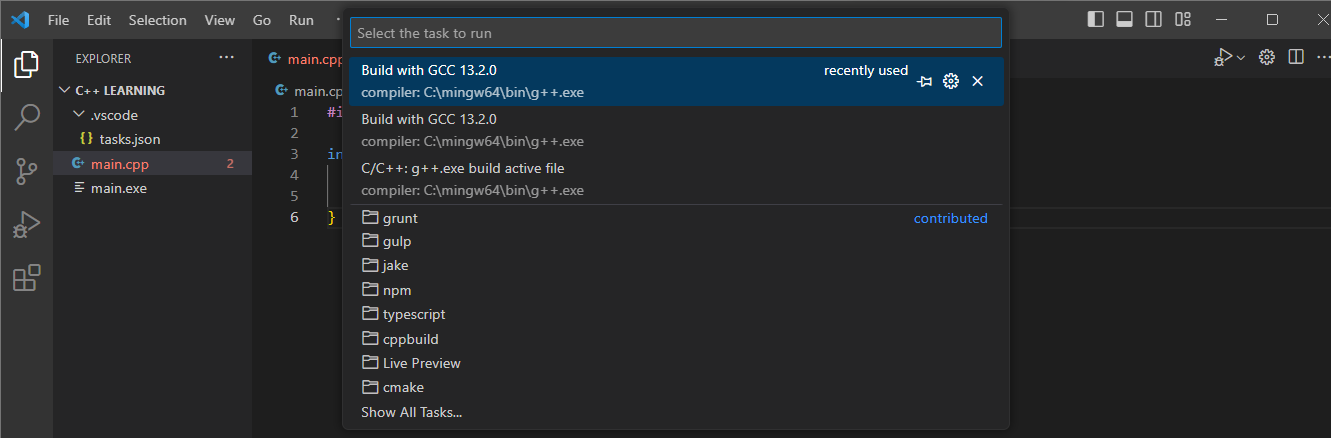
We can also check the tutorial from the C++ extension.

1. Open a new terminal -> Run task -> *main.exe* is created.

A screenshot of a computer

Description automatically generated

1. Select the compiler. (We can change the display name by modifying labels in the JSON file.)



1. If the IDE has been connected to our gcc compiler, we can type *.\main.exe* to run the program.

A black background with white text

Description automatically generated

* It is always good to have stored args so that we can directly copy and paste our configuration into each project's json. *(${workspaceFolder}\\\*.cpp* builds multiple files in the same project.)

"args": [

                "-fdiagnostics-color=always",

                "-g",

                "-std=c++20",

                "${workspaceFolder}\\\*.cpp",

                "-o",

                "${fileDirname}\\${fileBasenameNoExtension}.exe"

         ]

* How can we configure our project to compile it in another compiler?
  1. Terminal -> Configure Task -> Select the new compiler

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

* 1. The JSON file will have a new configuration for the new compiler.
  2. The args section can be the same as GCC.
* How to use the compiler from Microsoft (Msvc)
  1. Open Developer Command Prompt for VS
  2. *cd <project dir>*
  3. Open the project in VS code by command *code .*

A screenshot of a computer program

Description automatically generated

* Why do we open our project in this way?
  + Because the terminal can only identify Msvc in this situation as the compiler has not been added into PATH.

A screenshot of a computer

Description automatically generated

1. Terminal -> Configure Task -> C/C++: cl.exe build active file

A screenshot of a computer

Description automatically generated

1. Default args for Msvc json file:

"args": [

                "/Zi",

                "/std:c++latest",

                "/EHsc",

                "/Fe:",

                "${workspaceFolder}\\\*.cpp",

                "${fileDirname}\\${fileBasenameNoExtension}.exe"

         ]

* How to configure C/C++ extension in VS code?
* View -> Command Palette -> C/C++ Edit Configurations (UI)

A screenshot of a computer

Description automatically generated

**Chapter 3: Data Type**

**3.1 Basic Data Types**

|  |  |  |
| --- | --- | --- |
| **Data Type** | **Size** | **Description** |
| boolean | 1 byte | Stores true or false values |
| char | 1 byte | Stores a single character/letter/number, or ASCII values |
| int | 2 or 4 bytes | Stores whole numbers, without decimals |
| float | 4 bytes | Stores fractional numbers, containing one or more decimals. Sufficient for storing 6-7 decimal digits |
| double | 8 bytes | Stores fractional numbers, containing one or more decimals. Sufficient for storing 15 decimal digits |

**3.2 Constant**

* Make the variable **unchangeable and read-only.**
* E.g. **const** int myNum = 15;

**3.3 Auto**

* To let the compiler deduce the type.
* There are several suffixes to modify int and fractional number. (e.g., *f*, ***l***, ***u***, ***ul***, ***ll***)
* More examples can be found in Chapter 5.11.

**3.4 String**

* A string variable contains a collection of characters surrounded by double quotes.
* Two ways to concatenate different strings.
  + Concatenation
    - The ***+*** operator can be used between strings to add them together to make a new string. For example,

string firstName = "John ";  
string lastName = "Doe";  
string fullName = firstName + " " + lastName;

* + Append
    - A string in C++ is actually an object, which contain functions that can perform certain operations on strings. For example, you can also concatenate strings with the ***append()*** function. For example,

string firstName = "John ";  
string lastName = "Doe";  
string fullName = firstName.append(lastName);

* Length
  + To get the length of a string, use the ***length()*** or ***size()*** function, the following example string’s length is 26:

string txt = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";  
cout << "The length of the txt string is: " << txt.length();

cout << "The length of the txt string is: " << txt.size();

* Access/Slice
* The characters in a string by referring to its index number inside square brackets ***[]***. E.g.,

string myString = "Hello";  
cout << myString[0]; // prints the first character in myString

myString[0] = 'J'; // "Hello" -> "Jello"

* GetLine
  + To read a line of text. It takes cin as the first parameter, and the string variable as second.
  + Example:

string fullName;  
cout << "Type your full name: ";  
getline (cin, fullName);  
cout << "Your name is: " << fullName;  
  
// Type your full name: John Doe  
// Your name is: John Doe

* + More example checks Chapter 5.4 Inputs and Outputs

**Chapter 4: Operation**

**4.1 Basic Operations**

* add
* subtract
* divide
* modulus
* multiply

**Chapter 5: Code Examples**

**5.1 First Program**

#include <iostream>

// using namespace std;

consteval int get\_value(){

    return 3;

}

int main(){

    constexpr int value = get\_value();

    std::cout << "value : " << value << std::endl;

    return 0;

}

***#include <iostream>***: To include a standard library header file, which provides input and output functionality, such as ***std::cout*** and ***std::endl***.

***int main()***: The entry point of a C++ program.

***std::cout***: Print something.

***std::endl***: Print a new line. This can also be replaced by ***cout << "\n"***.

***using namespace std***: Can be used to omit ***std::***

***return 0***: Send message to the operation system if the program is finished successfully or if there is some kind of problem.

**5.2 Comments**

#include <iostream>

// Entry point main function

int main(int argc, char \*\*argv)

{

    //One line comment

    /\*

        Multi-line block comment

        Another line

        Oh! And another one!

    \*/

   //Print out some text

   std::cout << "Hello World in C++20!" << std::endl;

   return 0;

}

* What is the use of comments?
  + Comments can be used to explain C++ code, and to make it more readable.
  + It can also be used to prevent execution when testing alternative code.
  + The compiler will ignore comments.

**5.3 Functions**

#include <iostream>

// A function that adds two numbers and returns the result

int addNumbers(int first\_number, int second\_number){

    int sum = first\_number + second\_number;

    return sum;

}

int main(int argc, char \*\*argv){

    int firstNumber = 12;

    int secondNumber {9}; // Another way to initialize variables

    int sum = firstNumber + secondNumber;

    std::cout << "The sum of " << firstNumber << " and " << secondNumber <<" is " << sum << std::endl;

    std::cout << "The sum of " << firstNumber << " and " << secondNumber <<" is " << addNumbers(firstNumber, secondNumber) << std::endl;

    return 0;

}

***int addNumbers***: A function that adds two numbers and returns the result.

* A function must be defined before it is used.
* The advantage of function is reusable.

**5.4 Inputs and Outputs**

#include <iostream>

#include <string> // To use std::string

int main(int argc, char \*\*argv){

    // std::cout: Printing stuff to the console

    std::cout << "Hello World!" << std::endl;

    // Error

    std::cerr << "std::cerr output: Something went wrong." << std::endl;

    // Log message

    std::clog << "std::clog output: This is a log message." << std::endl;

    int age;

    std::string name;

    std::cout << "Please type in your Last Name: " << std::endl;

    // Input

    std::cin >> name;

    std::cout << "Please type in your age: " << std::endl;

    std::cin >> age;

    std::cout << "Hello " << name << "! You are " << age << " years old." << std::endl;

    // Chaining std::cin

    std::cout << "Please type in your Last Name and age, separated by spaces: " << std::endl;

    std::cin >> name >> age;

    std::cout << "Hello " << name << "! You are " << age << " years old." << std::endl;

    // Reading data with spaces

    std::string full\_name;

    std::cout << "Please type in your full name and age: " << std::endl;

    std::getline(std::cin, full\_name); // Enter then input age

    std::cin >> age;

    std::cout << "Hello " << full\_name << "! You are " << age << " years old." << std::endl;

    return 0;

}

***std::cout***: To print/output the variable to the console(terminal). The output is stored in a buffer until it is flushed. The buffer is typically flushed when a newline character (***\n***) is encountered or the program exits.

***std::cerr***: An unbuffered stream that is useful for printing error messages or other critical information that needs to be displayed immediately. The output is immediately sent to the console without being stored in a buffer.

***std::log***: Printing log messages to the console. It is buffered like ***cout***. This makes it useful for logging information that does not need to be displayed immediately, but still needs to be stored in a buffer for later retrieval.

***std::cin***: Reading data from the terminal. The example above shows three usages of ***cin***.

***#include <string>***: To include a standard library header file, which provides ***std::string***

**5.5 Number Systems**

#include <iostream>

int main(){

    int number1 = 15; // Decimal

    int number2 = 017; // Octal

    int number3 = 0x0F; // Hexadecimal

    int number4 = 0b00001111; // Binary

    std::cout << "number 1:" << number1 << std::endl;

    std::cout << "number 2:" << number2 << std::endl;

    std::cout << "number 3:" << number3 << std::endl;

    std::cout << "number 4:" << number4 << std::endl;

    return 0;

}

* The above program represents the same number in four different representations.
  + Decimal:
  + Octal:
  + Hexadecimal:
  + Binary:
* All data is represented by a bunch of grouped cells of 0’s and 1’s in memory.
* As the range of your data grows, so will the number of digits you need to represent the data in memory.
* Hexadecimal system makes it a little easier for humans to handle data streams with 1’s and 0’s.

N

* Octal has the same goal as Hexadecimal, but it’s almost no longer used in modern times. It is just mentioned here for awareness.

**5.6 Initialization**

#include <iostream>

int main(){

    // Braced Initialization

    int elephant\_count; // Variable may contain random garbage value. Warning

    int lion\_count{}; // Initialized to 0

    int dog\_count{10}; // Initializes to 10

    int cat\_count{15}; // Initializes to 15

    // Can use expression as initializer

    int domesticated\_animals{dog\_count + cat\_count};

    // int narrowing\_conversion{2.9}; // Warning: Some of the compiler will chop off the decimal part or throw an error

    std::cout << "Elephant count: " << elephant\_count << std::endl;

    std::cout << "Lions count: " << lion\_count << std::endl;

    std::cout << "Dogs count: " << dog\_count << std::endl;

    std::cout << "Cats count: " << cat\_count << std::endl;

    // Functional Initialization

    int apple\_count(5);

    int orange\_count(10);

    int fruit\_count{apple\_count + orange\_count};

    int narrowing\_conversion\_functional(2.9); // Information lost, 2 is stored instead of 2.9.

    std::cout << "Apples count: " << apple\_count << std::endl;

    std::cout << "Oranges count: " << orange\_count << std::endl;

    std::cout << "Fruits count: " << fruit\_count << std::endl;

    std::cout << "Narrowing conversion (Functional Initialization): " << narrowing\_conversion\_functional << std::endl;

    // Assignment Initialization

    int bike\_count = 2;

    int truck\_count = 7;

    int vehicle\_count = bike\_count + truck\_count;

    int narrowing\_conversion\_assignment = 2.9; // Information lost, 2 is stored instead of 2.9.

    std::cout << "Bikes count: " << bike\_count << std::endl;

    std::cout << "Trucks count: " << truck\_count << std::endl;

    std::cout << "Vehicles count: " << vehicle\_count << std::endl;

    std::cout << "Narrowing conversion (Assignment Initialization): " << narrowing\_conversion\_assignment << std::endl;

    // Check the size with sizeof

    std::cout << "sizeof int: " << sizeof(int) << std::endl; // 4 bytes

    std::cout << "sizeof truck\_count: " << sizeof(truck\_count) << std::endl; // 4 bytes

    return 0;

}

* There are three ways to initialize a variable.
  + Braced Initialization
  + Functional Initialization
  + Assignment Initialization

***sizeof***: Used to measure the memory size of a variable. For integer, it should be 4 bytes.

**5.7 Integer**

#include <iostream>

int main(){

    int value1{10};

    int value2{-300};

    std::cout << "value1: " << value1 << std::endl;

    std::cout << "value2: " << value2 << std::endl;

    std::cout << "sizeof(value1):" << sizeof(value1) << std::endl; // 4 bytes

    std::cout << "sizeof(value2):" << sizeof(value2) << std::endl; // 4 bytes

    std::cout << "---------------------" << std::endl;

    unsigned int value3{4};

    // unsigned int value4{-5}; // Compiler Error: negative value assigned to unsigned int

    // short and long

    short short\_var{-32768}; // 2 bytes

    short int short\_int{455};

    signed short signed\_short{122};

    signed short int signed\_short\_int{-456};

    unsigned short int unsigned\_short\_int{456};

    int int\_var {55} ; // 4 bytes

    signed signed\_var {66};//

    signed int signed\_int {77};//

    unsigned int unsigned\_int{77};

    long long\_var{88}; // 4 or 8 bytes

    long int long\_int{33};

    signed long signed\_long{44};

    signed long int signed\_long\_int{44};

    unsigned long int unsigned\_long\_int{44};

    long long long\_long{888}; // 8 bytes

    long long int long\_long\_int{999};

    signed long long signed\_long\_long{444};

    signed long long int signed\_long\_long\_int{1234};

    unsigned long long int unsigned\_long\_long\_int{1234};

    std::cout << "Short variable : " << short\_var<<  " , size : "

        << sizeof (short) << " bytes" << std::endl;

    std::cout << "Short Int : " << short\_int << " , size : "

    << sizeof (short int) << " bytes" << std::endl;

    std::cout << "Signed short : " << signed\_short

    << " , size : " << sizeof (signed short) << " bytes" << std::endl;

    std::cout << "Signed short int :  " << signed\_short\_int

    <<  " , size : " << sizeof (signed short int) << " bytes" << std::endl;

    std::cout << "unsigned short int :  " << unsigned\_short\_int

    <<  " , size : " << sizeof (unsigned short int) << " bytes" << std::endl;

    std::cout << "---------------------" << std::endl;

    std::cout << "Int variable :  " << int\_var <<" , size : "

        << sizeof (int) << " bytes" << std::endl;

    std::cout << "Signed variable " << signed\_var <<" , size : "

        << sizeof (signed) << " bytes" << std::endl;

    std::cout << "Signed int :  " << signed\_int <<" , size : "

        << sizeof (signed int) << " bytes" << std::endl;

    std::cout << "unsigned int :  " << unsigned\_int <<" , size : "

        << sizeof (unsigned int) << " bytes" << std::endl;

    std::cout << "---------------------" << std::endl;

    std::cout << "Long variable :  " << long\_var << " , size : "

        << sizeof (long) << " bytes" <<std::endl;

    std::cout << "Long int :  " << long\_int <<" , size : "

        << sizeof (long int) << " bytes" << std::endl;

    std::cout << "Signed long :  " << signed\_long <<" , size : "

        << sizeof (signed long) << " bytes" << std::endl;

    std::cout << "Signed long int : " << signed\_long\_int <<" , size : "

        << sizeof (signed long int) << " bytes" << std::endl;

    std::cout << "unsigned long int : " << unsigned\_long\_int <<" , size : "

        << sizeof (unsigned long int) << " bytes" << std::endl;

    std::cout << "---------------------" << std::endl;

    std::cout << "Long long : " << long\_long <<" , size : "

        << sizeof (long long) << " bytes" << std::endl;

    std::cout << "Long long int : " << long\_long\_int <<" , size : "

        << sizeof (long long int) << " bytes" << std::endl;

    std::cout << "Signed long long : " << signed\_long\_long <<" , size : "

        << sizeof (signed long long) << " bytes" <<std::endl;

    std::cout << "Signed long long int : " << signed\_long\_long\_int <<" , size : "

        << sizeof (signed long long int) << " bytes" << std::endl;

    std::cout << "unsigned long long int : " << unsigned\_long\_long\_int <<" , size : "

        << sizeof (unsigned long long int) << " bytes" << std::endl;

    std::cout << "---------------------" << std::endl;

    return 0;

}

|  |  |  |
| --- | --- | --- |
| **Type with modifier** | **Bytes in memory** | **Range** |
| Unsigned int | 4 |  |
| Signed int | 4 |  |

* is the number of bits for a type in memory.
* An unsigned integer can only store positive numbers.
* If the range of the stored value is larger than the range, we should consider use another type to store the value.
* Size of an integer with ***short*** type is 2 bytes.
* Size of an integer with ***long*** type is 4 bytes.
* Size of an integer with ***long long*** type is 8 bytes.

**5.8 Fractional Numbers**

#include <iostream>

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

int main(){

    // Declare and initialize the variables

    //Declare and initialize the variables

    float number1{1.12345678901234567890f};  // Size: 4bytes; Precision : 7

    double number2{1.12345678901234567890};  // Size: 8bytes; Precision : 15

    long double number3{1.12345678901234567890L}; // Size: 16bytes; Precision: 15+

    // Print out the sizes

    std::cout << "sizeof float : " << sizeof(float) << std::endl;

    std::cout << "sizeof double : " << sizeof(double) << std::endl;

    std::cout << "sizeof long double : " << sizeof(long double) << std::endl;

    //Precision

    std::cout << std::setprecision(20); // Control the precision from std::cout.

    std::cout << "number1 is : " << number1 << std::endl; //7 digits

    std::cout << "number2 is : " << number2 << std::endl; // 15'ish digits

    std::cout << "number3 is : " << number3 << std::endl; // 15+ digits

    //Float problems : The precision is usually too limited

    //for a lot of applications

    // float number4 {192400023.0f};               // Error : narrowing conversion, junk will be stored in number4

    double number4 {192400023.0};               // OK

    std::cout << "number4 : " << number4 << std::endl;

    //Scientific notation

    //What we have seen so far in terms of floating point types

    //is fixed notation. There is another notation, scientific

    //that is handy if you have really huge numbers or small numbers

    //to represent

    std::cout << "-------------------------" << std::endl;

    double number5 {192400023};

    double number6 {1.92400023e8};

    double number7 {1.924e8};           // Can ommit the lower 00023

                                        // for simplicity if our application allows that.

    double number8 {0.00000000003498};

    double number9 {3.498e-11}; // multiply with 10 exp(-11)

    std::cout << "number5 is : " << number5 << std::endl;

    std::cout << "number6 is : " << number6 << std::endl;

    std::cout << "number7 is : " << number7 << std::endl;

    std::cout << "number8 is : " << number8 << std::endl;

    std::cout << "number9 is : " << number9 << std::endl;

    //Infinity and Nan

    std::cout << std::endl;

    std::cout << "Infinity and NaN" << std::endl;

    double number10{ -5.6 };

    double number11{};//Initialized to 0

    double number12{};  //Initialized to 0

    //Infinity

    double result { number10 / number11 };

    std::cout << number10 << "/" << number11 << "  yields " << result << std::endl;

    std::cout << result << " + " << number10 << " yields " << result + number10 << std::endl;

    //NaN

    result = number11 / number12;

    std::cout << number11 << "/" << number12 << " = " << result << std::endl;

    return 0;

}

***std::setprecision***: Control the precision from std::cout. Need to ***#include <iomanip>***.

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Size** | **Precision** | **Comment** |
| float | 4 | 7 | - |
| double **(Default Type)** | 8 | 15 | Recommended default |
| long double | 12 | > double |  |

* Scientific Notation
* ***double number4 {2.5f}*** still equals ***float number 4 {2.5f}***. Hence, we need to remove ***f*** to convert it to double type: ***double number4 {2.5}***.
* Special Case

**5.9 Booleans**

#include <iostream>

int main(){

    bool red\_light{false};

    bool green\_light{true};

    if(red\_light == true){

        std::cout << "Stop!" << std::endl;

    }

    else{

        std::cout << "Go through!" << std::endl;

    }

    if(green\_light == true){

        std::cout << "The light is green!" << std::endl;

    }

    else{

        std::cout << "The light is NOT green!" << std::endl;

    }

    // sizeof()

    std::cout << "sizeof(bool): " << sizeof(bool) << std::endl; // 8 bytes

    //Printing out a bool

    //1 -->> true

    //0 -->> false

    std::cout << std::endl;

    std::cout << "red\_light : " << red\_light << std::endl; // 1

    std::cout << "green\_light : " << green\_light << std::endl; // 0

    std::cout << std::boolalpha; // Print out true or false instead of 1 or 0

    std::cout << "red\_light : " << red\_light << std::endl;

    std::cout << "green\_light : " << green\_light << std::endl;

    return 0;

}

* Boolean is often used to create condition statements.

***std::cout<<std::boolalpha***: Allows printing out true or false instead of 1 or 0.

**5.10 Characters and Text**

#include <iostream>

int main(){

    char character1 {'a'};

    char character2 {'r'};

    char character3 {'r'};

    char character4 {'o'};

    char character5 {'w'};

    std::cout << character1 << std::endl;

    std::cout << character2 << std::endl;

    std::cout << character3 << std::endl;

    std::cout << character4 << std::endl;

    std::cout << character5 << std::endl;

    std::cout << "sizeof(char): " << sizeof(char) << std::endl; // 1 byte (256 different values ( 0 - 255))

    char value = 65; // ASCII character code for 'A'

    std::cout << "value: " << value << std::endl; // print A

    std::cout << "value(int): " << static\_cast<int>(value) << std::endl; // print 65

    return 0;

}

***static\_cast<int>(char)***: Convert char variable to int. Usually will be converted to ASCII.

**5.11 Auto**

#include <iostream>

int main(){

    auto var1{12}; // int

    auto var2{13.0}; // double

    auto var3{14.0f}; // float

    auto var4{15.01l}; // long double

    auto var5{'e'}; // char

    // int modifier suffixes

    auto var6{123u};    // unsigned

    auto var7{123ul};   // unsigned long

    auto var8{123ll};   // long long

    std::cout << "var1 occupies : " << sizeof(var1) << " bytes" << std::endl; // 4 bytes

    std::cout << "var2 occupies : " << sizeof(var2) << " bytes" << std::endl; // 8 bytes

    std::cout << "var3 occupies : " << sizeof(var3) << " bytes" << std::endl; // 4 bytes

    std::cout << "var4 occupies : " << sizeof(var4) << " bytes" << std::endl; // 16 bytes

    std::cout << "var5 occupies : " << sizeof(var5) << " bytes" << std::endl; // 1 byte

    std::cout << "var6 occupies : " << sizeof(var6) << " bytes" << std::endl; // 4 bytes

    std::cout << "var7 occupies : " << sizeof(var7) << " bytes" << std::endl; // 4 bytes

    std::cout << "var8 occupies : " << sizeof(var8) << " bytes" << std::endl; // 8 bytes

    return 0;

}

* To let the compiler deduce the type.
* There are several suffixes to modify int and fractional number. (e.g., *f*, ***l***, ***u***, ***ul***, ***ll***)

**5.12 Data Assignment**

#include <iostream>

int main(){

    int var1{123}; // Declare and initialize

    std::cout << "var1 : "  << var1 << std::endl;

    var1 = 55; // Assign

    std::cout << "var1 : "  << var1 << std::endl;

    std::cout << "----------------" << std::endl;

    double var2 {44.55}; // Declare and initialize

    std::cout << "var2 : " << var2 << std::endl;

    var2 = 99.99; // Assign

    std::cout << "var2 : " << var2 << std::endl;

    std::cout << "----------------" << std::endl;

    bool state{false}; // Declare and initialize

    std::cout << std::boolalpha;

    std::cout << "state : " << state << std::endl;

    state = true; // Assign

    std::cout << "state : " << state << std::endl;

    //Auto type deduction

    //Careful about auto assignments

    auto var3 {333u}; // Declare and initialize with type deduction

    var3 = -22; // Assign negative number. DANGER! (Junk will be assigned instead)

    std::cout << "var3 : " << var3 << std::endl;

    return 0;

}

* Be careful about **assignments to auto variable**, the assignment should not be different type of the auto initialized data type. Otherwise, a random number will be assigned to the variable instead.

**5.13 Basic Operations**

#include <iostream>

int main(){

    //Addition

    int number1{2};

    int number2{7};

    int result = number1 + number2;

    std::cout << "result : " << result << std::endl; // 9

    //Subtraction

    result = number2 - number1 ;

    std::cout << "result : " << result << std::endl; // 5

    result = number1 - number2;

    std::cout << "result : " << result << std::endl; // -5

    //Multiplication

    result = number1 \* number2;

    std::cout << "result : " << result << std::endl; // 14

    //Division

    result = number2 / number1;

    std::cout << "result : " << result << std::endl; // 3

    //Modulus

    result = number2 % number1; // 7 % 2

    std::cout << "result : " << result << std::endl; // 1

    result = 31 % 10;

    std::cout << "result : " << result << std::endl; // 1

    return 0;

}

* Integer Division ***/*** can only get Integer.
* Modulus ***%*** returns the remainder of the division.

**5.14 Precedence and Associativity**

#include <iostream>

int main(){

    int a {6};

    int b {3};

    int c {8};

    int d {9};

    int e {3};

    int f {2};

    int g {5};

    int result = a + b \* c -d/e -f + g; //  6 +  (3\*8)  -   (9/3) - 2 + 5

    std::cout << "result : " << result << std::endl; // 30

    result = a/b\*c +d - e + f;  //   (6/3)\*8 + 9 - 3 + 2

    std::cout << "result : " << result << std::endl; // 24

    result = (a + b) \* c -d/e -f + g; // (6+3)\*8 -(9/3) -2 + 5

    std::cout << "result () : " << result << std::endl; // 72

    return 0;

}

* A rule to follow when multiple operations exist in the same expression.
  + Precedence: which operation to do first.
  + Associativity: which direction or which order.
* It is better to make the intent in code as clear as possible by clearly using ***()*** to indicate which operations should be done first.
* C++ operator precedence table: <https://en.cppreference.com/w/cpp/language/operator_precedence>

**5.15 Prefix and Postfix + and -**

#include <iostream>

int main(){

    int value { 5 };

    //Increment by one

    value = value + 1; //6

    std::cout << "The value is : " << value << std::endl; // 6

    value = 5; // Reset value to 5

    //Decrement by one

    value = value - 1; // 4

    std::cout << "The value is : " << value << std::endl; //4

    //===================================================================

    std::cout << "======Postfix increment and decrement======"<< std::endl;

    //Reset value to 5

    value = 5;

    std::cout << "The value is (incrementing) : " << value++ << std::endl; // 5

    std::cout << "The value is : " << value << std::endl; // 6

    std::cout << std::endl;

    //Decrement with postfix

    //Reset value to 5

    value = 5;

    std::cout << "The value is (decrementing) : " << value-- << std::endl; //5

    std::cout << "The value is : " << value << std::endl; // 4

    //===================================================================

    std::cout << "======Prefix increment and decrement======"<< std::endl;

    //Reset value to 5

    value = 5;

    ++value;

    std::cout << "The value is (prefix++) : " << value << std::endl; // 6

    //Reset value to 5

    value = 5;

    std::cout << "The value is (prefix++ in place) : " << ++value << std::endl; // 6

    std::cout << std::endl;

    //Prefix : Decrementing

    //Reset value to 5;

    value = 5;

    --value;

    std::cout << "The value is (prefix--) : " << value << std::endl; // 4

    //Reset value to 5;

    value = 5;

    std::cout << "The value is (prefix-- in place) : " << --value << std::endl;//4

    return 0;

}

* Prefix and postfix increment/decrement operators can only increment/decrement by 1.
* Prefix and postfix operators are only available for + and -.
* Difference between the prefix and postfix operator is the execution order. For example,
  + ***a++***: Returns ***a*** and change ***a = a + 1***. (Postfix)
  + ***++a***: Returns ***a + 1*** and change ***a = a + 1***. (Prefix)

**References**

* [**C++ Programming Course - Beginner to Advanced**](https://www.google.com/url?q=https://www.youtube.com/watch?v%3D8jLOx1hD3_o%26ab_channel%3DfreeCodeCamp.org&sa=D&source=calendar&usd=2&usg=AOvVaw32IHq1-uzixQnuCtxAm8fA)
* [**W3schools C++ Tutorial**](https://www.w3schools.com/cpp/cpp_intro.asp)