In this research, we will follow some roadmaps to learn C++ aka cpp/cplusplus. I will kin on learning how to make programming languages and their compiler etc. I will mention some websites through which any person can learn C++.

# ROADMAP

# INTRODUCTION

* Compilers

# BASICS OF C++

* Structure of a program
* Variables and types
* Constants
* Operators
* Basic Input/Output

# PROGRAM STRUCTURE

* Control structures
* Functions
* Overloads and templates
* Name visibility

# Compound data types

* Arrays
* Character sequences
* Pointers – it’s a myth that they are difficult but they are quite useful
* Dynamic Memory
* Data structures
* Other data types

# Classes

* Classes (I)
* Classes (II)
* Special members
* Friendship and inheritance
* Polymorphism

# OTHER LANGUAGES FEATURES

* Type conversions
* Exceptions
* Preprocessor directives

# C++ STANDARD LIBRARY

* Input/Output with files

Let’s begin with the introduction with compilers.

FAQ: How to a make compiler?

[c - How to write a very basic compiler - Software Engineering Stack Exchange](https://softwareengineering.stackexchange.com/questions/165543/how-to-write-a-very-basic-compiler#:~:text=A%20typical%20compiler%20does%20the%20following%20steps%3A%20Parsing%3A,no%20sense%2C%20e.g.%20unreachable%20code%20or%20duplicate%20declarations.)

# INTRODUCTION

# COMPILERS

Compilers

The essential tools needed to follow these tutorials are a computer and a compiler toolchain able to compile C++ code and build the programs to run on it.

C++ is a language that has evolved much over the years, and these tutorials explain many features added recently to the language. Therefore, in order to properly follow the tutorials, a recent compiler is needed. It shall support (even if only partially) the features introduced by the 2011 standard.

Many compiler vendors support the new features at different degrees. See the bottom of this page for some compilers that are known to support the features needed. Some of them are free!

If for some reason, you need to use some older compiler, you can access an older version of these tutorials here (no longer updated).

[Recommended Compiler for C++] <https://www.mingw-w64.org/downloads/#msys2>

Two simple commands to compile cpp code, g++ [name file] -o [executable name you want].

What is a compiler?

Computers understand only one language and that language consists of sets of instructions made of ones and zeros. This computer language is appropriately called machine language.

A single instruction to a computer could look like this:

|  |  |
| --- | --- |
| 000000 | 1010101010101 |

A particular computer’s machine language program that allows a user to input two numbers, adds the two numbers together, and displays the total could include these machine code instructions:

|  |  |
| --- | --- |
| 00000 | 10101000 |
| 00001 | 11111000 |
| 00010 | 10001110 |

Etc.

As you can imagine, programming a computer directly in machine language using only ones and zeros is very tedious and error prone. To make programming easier, higher-level languages have been developed. High level programs also make it easier for programmers to inspect and understand each other’s programs easier.

This is a portion of code written in C++ that accomplishes the exact same purpose:

1. int a, b, sum;
2. cin>>a;
3. cin>>b;
4. sum=a+b;
5. cout<<sum<<endl;

even if you cannot really understand the code above, you should be able to appreciate how much easier it will be to program in the C++ language as opposed to machine language.

Because a computer can only understand machine language and humans wish to write in high level languages high level languages have to be re-written (translated) into machine language at some point. This is done by special programs called compilers, interpreters, or assemblers that are built into the various programming applications.

C++ is designed to be a compiled language, meaning that is a generally translated into machine language that can be understood directly by the system, making the generated program highly efficient. For that, a set of tools are needed, known as the development toolchain, whose core are a compiler and its linker.

Console programs

Console programs are programs that use text to communicate with the user and the environment, such as printing text to the screen or reading input from a keyboard.

Console programs are easy to interact with, and generally have a predictable behavior that is identical across all platforms. They are also simple to implement and thus are very useful to learn the basics of a programming languages:

The examples in these tutorials are all console programs.

The way to compile console programs depends on the particular tool you are using.

The easiest way for beginners to compile C++ programs is by using an Integrated Development Environment (IDE). An IDE generally integrates several development tools, including a text editor and tools to compile programs directly from it.

Here you have instructions on how to compile and run console programs using different free Integrated Development Interfaces (IDEs):

|  |  |  |
| --- | --- | --- |
| **IDE** | **Platform** | **Console programs** |
| **Code::blocks** | Windows/Linux/MacOS | [Compile console programs using Code::blocks](https://www.cplusplus.com/doc/tutorial/introduction/codeblocks/) |
| **Visual Studio Express** | Windows | [Compile console programs using VS Express 2013](https://www.cplusplus.com/doc/tutorial/introduction/visualstudio/) |
| **Dev-C++** | Windows | [Compile console programs using Dev-C++](https://www.cplusplus.com/doc/tutorial/introduction/devcpp/) |

If you happen to have a Linux or Mac environment with development features, you should be able to compile any of the examples directly from a terminal just by including C++11 flags in the command for the compiler:

|  |  |  |
| --- | --- | --- |
| **Compiler** | **Platform** | **Command** |
| **GCC** | Linux, among others... | g++ -std=c++0x example.cpp -o example\_program |
| **Clang** | OS X, among others... | clang++ -std=c++11 -stdlib=libc++ example.cpp -o example\_program |

See [2] for more.

As per my research, Visual studio code is suitable for coding because it supports document formatting, internal compiler extension etc.

# Basics of C++

**Structure of a program**

The best way to learn a programming language is by writing programs. Typically, the first program beginners write is a program called “Hello World!”, which simply prints “Hello World” to your computer screen. Although it is very simple, it contains all the fundamental components C++ programs have:

|  |  |
| --- | --- |
| 1 //my first program in C++  2 #include<iostream>  3  4 int main ()  5{   1. std :: count<<”Hello World!”; 2. } | Hello World! |

See [4] for more.

**Variables and types**

The usefulness of the “Hello World” programs shown in the previous chapter is rather questionable. We had to write several lines of code, compile them, and then execute the resulting program, just to obtain the result of a simple sentence written on the screen. It certainly would have been much faster to type the output sentence ourselves.

However, programming is not limited only to printing simple texts on the screen. In order to go a little further on and to become able to write programs that perform useful tasks that really save us work, we need to introduce the concept of variables.

**Identifiers**

A *valid identifier* is a sequence of one or more letters, digits, or underscore characters (\_). Spaces, punctuation marks, and symbols cannot be part of an identifier. In addition, identifiers shall always begin with a letter. They can also begin with an underline character (\_), but such identifiers are -on most cases- considered reserved for compiler-specific keywords or external identifiers, as well as identifiers containing two successive underscore characters anywhere. In no case can they begin with a digit.  
C++ uses a number of keywords to identify operations and data descriptions; therefore, identifiers created by a programmer cannot match these keywords. The standard reserved keywords that cannot be used for programmer created identifiers are:  
alignas, alignof, and, and\_eq, asm, auto, bitand, bitor, bool, break, case, catch, char, char16\_t, char32\_t, class, compl, const, constexpr, const\_cast, continue, decltype, default, delete, do, double, dynamic\_cast, else, enum, explicit, export, extern, false, float, for, friend, goto, if, inline, int, long, mutable, namespace, new, noexcept, not, not\_eq, nullptr, operator, or, or\_eq, private, protected, public, register, reinterpret\_cast, return, short, signed, sizeof, static, static\_assert, static\_cast, struct, switch, template, this, thread\_local, throw, true, try, typedef, typeid, typename, union, unsigned, using, virtual, void, volatile, wchar\_t, while, xor, xor\_eq  
Specific compilers may also have additional specific reserved keywords.  
**Very important:** The C++ language is a "case sensitive" language. That means that an identifier written in capital letters is not equivalent to another one with the same name but written in small letters. Thus, for example, the RESULT variable is not the same as the result variable or the Result variable. These are three different identifiers identifiying three different variables.

**Fundamental data types**

The values of variables are stored somewhere in an unspecified location in the computer memory as zeros and ones. Our program does not need to know the exact location where a variable is stored; it can simply refer to it by its name.

What the program needs to be aware of is the kind of data stored in the variable. It is not the same to store a simple integer as it is to store a letter or a large floating – point number; even though they are all represented using zeros and ones, they are not interpreted in the same way, and in many cases, they don’t occupy the same amount of memory.

Fundamental data types are basic types implemented directly by the language that represent the basic storage units supported natively by most systems. They can mainly be classified into:

* **Character types:**

They can represent a single character, such as ‘A’ or ‘$’. The most basic type is char, which is a one – byte character. Other types are also provided for wider characters.

* **Numerical integer types:**

They can store a whole number value, such as 7 or 1024. They exist in a variety of sizes, and can either be signed or unsigned, depending on whether they support negative values or not.

* **Floating – point types:**

They can represent real values, such as 3.14 or 0.01, with different levels of precision, depending on which of the three floating – point types is used.

* **Boolean type:**

The boolean type, known in C++ as bool, can only represent one of two states, true or false.

Here is the complete list of fundamental types in C++:

|  |  |  |
| --- | --- | --- |
| Group | Type names\* | Notes on size / precision |
| Character types | char | Exactly one byte in size. At least 8 bits. |
| char16\_t | Not smaller than char. At least 16 bits. |
| char32\_t | Not smaller than char16\_t. At least 32 bits. |
| wchar\_t | Can represent the largest supported character set. |
| Integer types (signed) | signed char | Same size as char. At least 8 bits. |
| *signed* short *int* | Not smaller than char. At least 16 bits. |
| *signed* int | Not smaller than short. At least 16 bits. |
| *signed* long *int* | Not smaller than int. At least 32 bits. |
| *signed* long long *int* | Not smaller than long. At least 64 bits. |
| Integer types (unsigned) | unsigned char | (same size as their signed counterparts) |
| unsigned short *int* |
| unsigned *int* |
| unsigned long *int* |
| unsigned long long *int* |
| Floating-point types | float |  |
| double | Precision not less than float |
| long double | Precision not less than double |
| Boolean type | bool |  |
| Void type | void | no storage |
| Null pointer | decltype(nullptr) |  |

\* The names of certain integer types can be abbreviated without their signed and int components - only the part not in italics is required to identify the type, the part in italics is optional. I.e., *signed* short *int* can be abbreviated as signed short, short int, or simply short; they all identify the same fundamental type.  
  
Within each of the groups above, the difference between types is only their size (i.e., how much they occupy in memory): the first type in each group is the smallest, and the last is the largest, with each type being at least as large as the one preceding it in the same group. Other than that, the types in a group have the same properties.  
  
Note in the panel above that other than char (which has a size of exactly one byte), none of the fundamental types has a standard size specified (but a minimum size, at most). Therefore, the type is not required (and in many cases is not) exactly this minimum size. This does not mean that these types are of an undetermined size, but that there is no standard size across all compilers and machines; each compiler implementation may specify the sizes for these types that fit the best the architecture where the program is going to run. This rather generic size specification for types gives the C++ language a lot of flexibility to be adapted to work optimally in all kinds of platforms, both present and future.  
  
Type sizes above are expressed in bits; the more bits a type has, the more distinct values it can represent, but at the same time, also consumes more space in memory:

|  |  |  |
| --- | --- | --- |
| Size | Unique representable values | Notes |
| 8-bit | 256 | = 28 |
| 16-bit | 65 536 | = 216 |
| 32-bit | 4 294 967 296 | = 232 (~4 billion) |
| 64-bit | 18 446 744 073 709 551 616 | = 264 (~18 billion billion) |

For integer types, having more representable values means that the range of values they can represent is greater; for example, a 16-bit unsigned integer would be able to represent 65536 distinct values in the range 0 to 65535, while its signed counterpart would be able to represent, on most cases, values between -32768 and 32767. Note that the range of positive values is approximately halved in signed types compared to unsigned types, due to the fact that one of the 16 bits is used for the sign; this is a relatively modest difference in range, and seldom justifies the use of unsigned types based purely on the range of positive values they can represent.  
For floating-point types, the size affects their precision, by having more or less bits for their significant and exponent.  
If the size or precision of the type is not a concern, then char, int, and double are typically selected to represent characters, integers, and floating-point values, respectively. The other types in their respective groups are only used in very particular cases.  
The properties of fundamental types in a particular system and compiler implementation can be obtained by using the [numeric\_limits](https://www.cplusplus.com/numeric_limits) classes (see standard header [<limits>](https://www.cplusplus.com/%3Climits%3E)). If for some reason, types of specific sizes are needed, the library defines certain fixed-size type aliases in header [<cstdint>](https://www.cplusplus.com/%3Ccstdint%3E).  
The types described above (characters, integers, floating-point, and boolean) are collectively known as arithmetic types. But two additional fundamental types exist: void, which identifies the lack of type; and the type nullptr, which is a special type of pointer. Both types will be discussed further in a coming chapter about pointers.  
C++ supports a wide variety of types based on the fundamental types discussed above; these other types are known as *compound data types*, and are one of the main strengths of the C++ language. We will also see them in more detail in future chapters. See [4] for more.

# REFERENCES

[1] [C++ Language - C++ Tutorials (cplusplus.com)](https://www.cplusplus.com/doc/tutorial/)

[2] [Compilers - C++ Tutorials (cplusplus.com)](https://www.cplusplus.com/doc/tutorial/introduction/)

[3] [Structure of a program - C++ Tutorials (cplusplus.com)](https://www.cplusplus.com/doc/tutorial/program_structure/)

[4] [Variables and types - C++ Tutorials (cplusplus.com)](https://www.cplusplus.com/doc/tutorial/variables/)