Software Documentation

Carmine Tommaso Recchiuto

Software Documentation

Writing a reliable documentation is a must for any programmer.

The presence of documentation helps keep track of all aspects of an application and it improves on the quality of a software product.

A good software documentation is fundamental for:

- development
- maintenance
- knowledge transfer to other developers

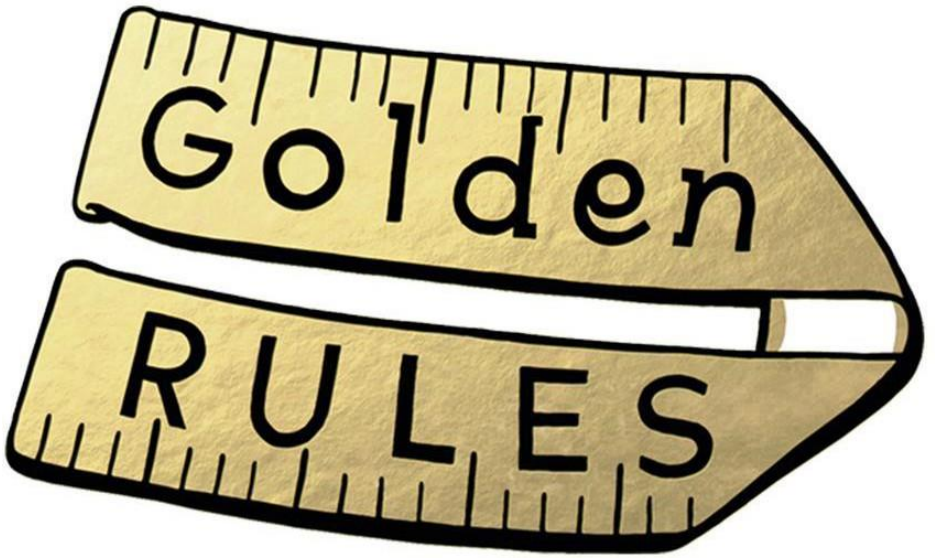
A successful documentation will make information easily accessible, help new users learn quickly, simplify the product and help cut support costs (if any!).

Software Documentation

- ✓ For all these reasons, you may guess that Software documentation is a critical activity in engineering, which may drastically improve the quality of a software product.
- ✓ Systematic approaches to documentation increase the level of confidence of the end deliverable as well as enhance and ensure product's success through its usability, marketability and ease of support
- ✓ Documentation is an activity that needs to commence early in development and continue throughout the development lifecycle. It acts as a tool for planning and decision making.

Seven Golden Rules

1. Documentation should be written from the point of view of the reader, not the writer.
2. Avoid repetition.
3. Avoid unintentional ambiguity.
4. Use a standard organization.
5. Record rationale
6. Keep it current
7. Review documentation for fitness of purpose



F. Bachmann, L. Bass, J. Carriere, P. Clements, D. Garlan, J. Ivers, R. Nord, R. Little, Software Architecture Documentation in Practice: Documenting Architectural Layers. CMU/SEI-2000- SR-004. Carnegie. Mellon University. 2000.

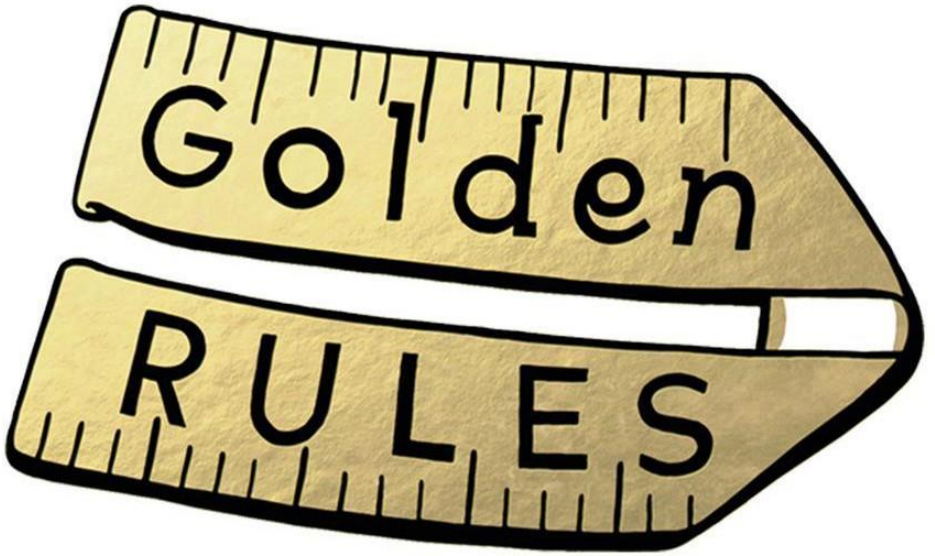
Other four golden rules

Keep comments as close to the code being described as possible. Comments that aren't near their describing code are frustrating to the reader and easily missed when updates are made.

Don't use complex formatting (such as tables or ASCII figures). Complex formatting leads to distracting content and can be difficult to maintain over time.

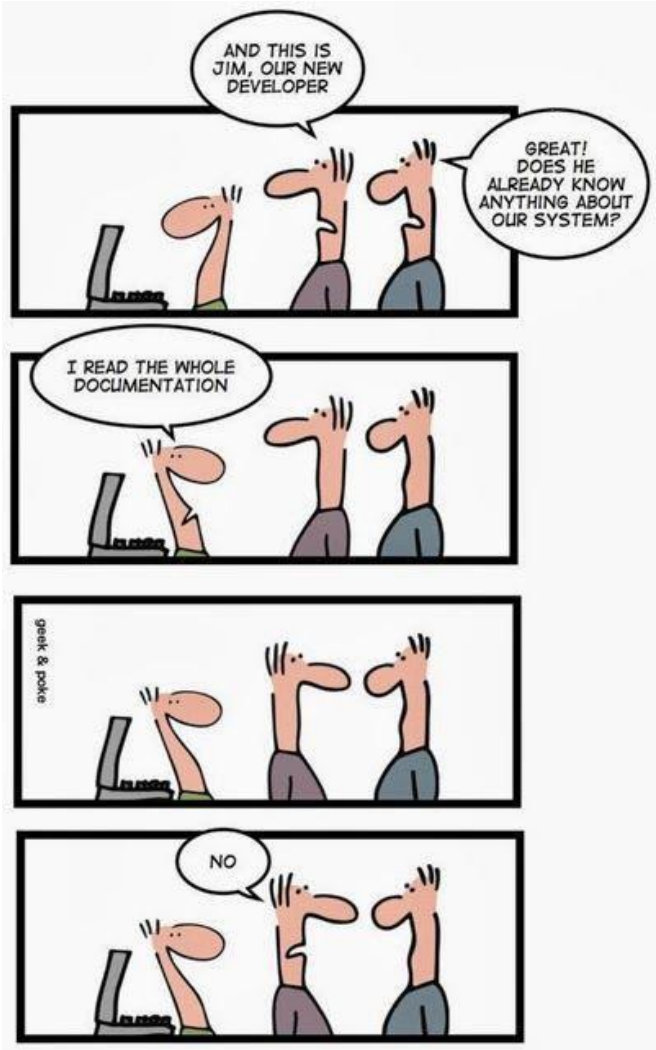
Don't include redundant information. Assume the reader of the code has a basic understanding of programming principles and language syntax.

Design your code to comment itself. The easiest way to understand code is by reading it. When you design your code using clear, easy-to-understand concepts, the reader will be able to quickly conceptualize your intent.



Documentation?

- ✓ API (functions, methods) documentation – Reference documentation regarding making calls and classes
- ✓ README – A high-level overview of the software, usually alongside the source code
- ✓ Release notes - Information describing the latest software or feature releases, and bug fixes
- ✓ System documentation – Documents describing the software system, including technical design documents, software requirements, and UML diagrams



Examples –Python

Comments should have a maximum length of 72 characters. This is true even if your project changes the max line length to be greater than the recommended 80 characters. If a comment is going to be greater than the comment char limit, using multiple lines for the comment is appropriate.

Planning and Reviewing: When you are developing new portions of your code, it may be appropriate to first use comments as a way of planning or outlining that section of code. Remember to remove these comments once the actual coding has been implemented and reviewed/tested:

 # First step
 # Second step
 # Third step

Examples –Python

Code Description: Comments can be used to explain the intent of specific sections of code:

 # Attempt a connection based on previous settings. If unsuccessful,
 # prompt user for new settings.

Algorithmic Description: When algorithms are used, especially complicated ones, it can be useful to explain how the algorithm works or how it's implemented within your code. It may also be appropriate to describe why a specific algorithm was selected over another.

 # Using quick sort for performance gains

Tagging: The use of tagging can be used to label specific sections of code where known issues or areas of improvement are located. Some examples are: BUG, FIXME, and TODO.

 # TODO: Add condition for when val is None

Examples – Docstrings in Python

Documenting your Python code is all centered on docstrings. These are built-in strings that, when configured correctly, can help your users and yourself with your project's documentation. They are stored in the property __doc__:

```
def say_hello(name):
    print("Hello " + name + " is it me you're looking for?")

say_hello.__doc__= "A simple function that says hello! "
```

Now try opening a python shell and typing

```
$ from test_docstrings import say_hello
$ help(say_hello)
```

Examples – Docstrings in Python

Python has one more feature that simplifies docstring creation. Instead of directly manipulating the __doc__property, the strategic placement of the string literal directly below the object will automatically set the __doc__value.

```
def say_hello(name):
    """A simple function that says hello... """
    print("Hello " + name + " is it me you're looking for?")
```

Examples – Docstrings in Python

Docstring conventions are described at https://www.python.org/dev/peps/pep-0257/ . Their purpose is to provide your users with a brief overview of the object. They should be kept concise enough to be easy to maintain but still be elaborate enough for new users to understand their purpose and how to use the documented object.

In all cases, the docstrings should use the triple-double quote (""") string format. This should be done whether the docstring is multi-lined or not. At a bare minimum, a docstring should be a quick summary of whatever is it you're describing and should be contained within a single line

"""This is a quick summary line used as a description of the object."""“

Multi-lined docstrings are used to further elaborate on the object beyond the summary. All multi-lined docstrings have the following parts:

- A one-line summary line
- A blank line proceeding the summary
- Any further elaboration for the docstring
- Another blank line

Examples – Docstrings in Python

"""This is the summary line

This is the further elaboration of the docstring. Within this section,
you can elaborate further on details as appropriate for the situation.
Notice that the summary and the elaboration is separated by a blank new line.
"""

Docstrings can be further broken up into three major categories:

- **Class Docstrings:** Class and class methods
- **Script Docstrings:** Scripts and functions

Python Documentation

The way you document your project should suit your specific situation. Keep in mind who the users of your project are going to be and adapt to their needs. Depending on the project type, certain aspects of documentation are recommended. The general layout of the project and its documentation should be as follows:

```
project_root/
|
├── project/  # Project source code
├── docs/
├── README
├── HOW_TO_CONTRIBUTE
├── CODE_OF_CONDUCT
├── examples.py
```

Python Documentation

Readme: A brief summary of the project and its purpose. Include any special requirements for installing or operating the projects. Additionally, add any major changes since the previous version. Finally, add links to further documentation, bug reporting, and any other important information for the project. Dan Bader has put together a great tutorial on what all should be included in your readme: https://dbader.org/blog/write-a-great-readme-for-your-github-project

How to Contribute: This should include how new contributors to the project can help. This includes developing new features, fixing known issues, adding documentation, adding new tests, or reporting issues.

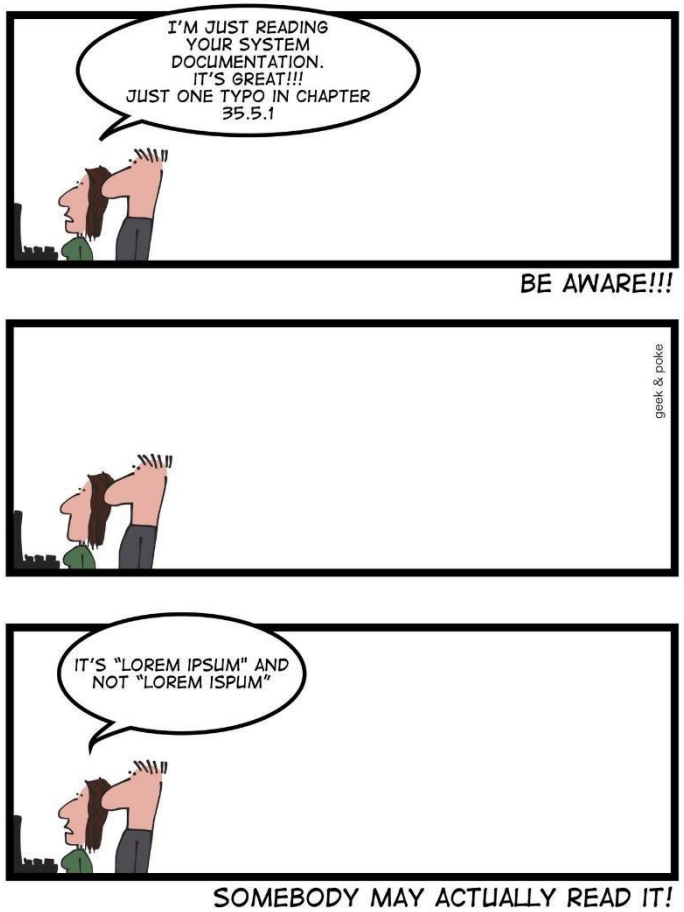
Code of Conduct: Defines how other contributors should treat each other when developing or using your software. This also states what will happen if this code is broken. If you're using Github, a Code of Conduct template can be generated with recommended wording. For Open Source projects especially, consider adding this.

Documentation Tools

Documenting your code, especially large projects, can be daunting. Thankfully there are some tools out and references to get you started:

Sphinx: https://www.sphinx-doc.org/en/master/ A collections of tools to auto-generate documentation in multiple formats

Doxygen: https://www.doxygen.nl/manual/docblocks.html A tool for generating documentation that supports Python as well as multiple other languages



Markdown Format

Markdown is intended to be as easy-to-read and easy-to-write as is feasible.

Readability, however, is emphasized above all else. A Markdown-formatted document should be publishable as-is, as plain text, without looking like it's been marked up with tags or formatting instructions. While Markdown's syntax has been influenced by several existing text-to-HTML filters, the single biggest source of inspiration for Markdown's syntax is the format of plain text email.

You can use Markdown most places around GitHub:

Comments in Issues and Pull Requests
Files with the .md or .markdown extension (README!)

Markdown Format

Github Flavored Markdown: dialect of Markdown that is currently supported for user content on GitHub.com and GitHub Enterprise.

https://github.github.com/gfm/#what-is-github-flavored-markdown-

Here's an overview of Markdown syntax that you can use anywhere on GitHub.com or in your own text files.

https://docs.github.com/en/github/writing-on-github/basic-writing-and-formatting-syntax

C++ (for Robotics)

Carmine Tommaso Recchiuto, Phd

C++

Along with Python, C++ is one of the most popular languages in for programming robots.

Of course we will not go into the details of C++, but if you are interesting there are very good tutorials over there.

Example:

https://www.geeksforgeeks.org/c-plus-plus/

Installing Compiler

C++ is a compiled language. What that means is that when you want to run a program that you write in C++, your machine needs to have a special program that translates the C++ that you write into language that the computer understands and can execute.

The built-in compiler for C/C++ (i.e. C language and C++ language) is called GCC/G++

The Docker image of the course already has the gcc container.

If you need to install it, use (On Linux):

apt install build-essential

Writing a program in C++

Let’s create a new folder on our workspace, and use gedit for writing some text.

Let’s write the following code:

```
//Simple C++ program to display Hello World
#include <iostream>
using namespace std;

int main()
{
   cout <<"Hello World“<<endl;
   return 0;
}
```

Writing a program in C++

//Simple C++ program to display Hello World

Comment. Any line beginning with '//' without quotes OR in between /*…*/ in C++ is comment.

#include <iostream>

In C++, all lines that start with pound (#) sign are called directives and are processed by preprocessor which is a program invoked by the compiler. The **#include** directive tells the compiler to include a file and **#include<iostream>** tells the compiler to include the standard iostream file which contains declarations of all the standard input/output library functions.

Writing a program in C++

using namespace std;

This is used to import the entirety of the std namespace into the current namespace of the program. The statement *using namespace std* is generally considered a bad practice. When we import a namespace we are essentially pulling all type definitions into the current scope. The alternative to this statement is to specify the namespace to which the identifier belongs using the scope operator(::) each time we declare a type.

```
//Simple C++ program to display Hello World
#include <iostream>

int main()
{

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

return 0;
}
```

The code on the left is equivalent to the one shown before, but without importing any namespace, which may led to ambiguities.

Writing a program in C++

int main () { }

This line is used to declare a function named “main” which returns data of integer type. A function is a group of statements that are designed to perform a specific task. Execution of every C++ program begins with the main() function, no matter where the function is located in the program. So, every C++ program must have a main() function.
The opening braces ‘{‘ indicates the beginning of the main function and the closing braces ‘}’ indicates the ending of the main function. Everything between these two comprises the body of the main function.

std::cout<<“Hello World”<< std:: endl;: This line tells the compiler to display the message “Hello World” on the screen. This line is called a statement in C++. Every statement is meant to perform some task. A semi-colon ‘;’ is used to end a statement. Semi-colon character at the end of statement is used to indicate that the statement is ending there. The cout is used to identify the standard character output device which is usually the desktop screen. Everything followed by the character “<<” is displayed to the output device. **endl** is used to let the terminal move to the next line.

Writing a program in C++

return 0;

This is also a statement. This statement is used to return a value from a function and indicates the finishing of a function. This statement is basically used in functions to return the results of the operations performed by a function.

Indentation: As you can see the cout and the return statement have been indented or moved to the right side. This is done to make the code more readable. In a program as Hello World, it does not hold much relevance seems but as the programs become more complex, it makes the code more readable, less error-prone. Therefore, you can always use indentations and comments to make the code more readable. However, differently from what you have seen in Python, here indentation is not mandatory and it is not used to define functions.

Building a program in C++

On the terminal, and in the work directory:

g++ hello.cpp

The a.out file has been generated, and you may notice that it is an executable file. We can run it by executing:

./a.out

It is also possible to set the name of the executable:

g++ hello.cpp –o hello

Statements if, for, while

If-else statement

```
if (x > y)
    x=y;
  else{
    x=x+1;
    y=y-1;
  }
```

for statement

```
for (int i = 0; i < 5; i++){
   std::cout << i << std::endl;
}
```

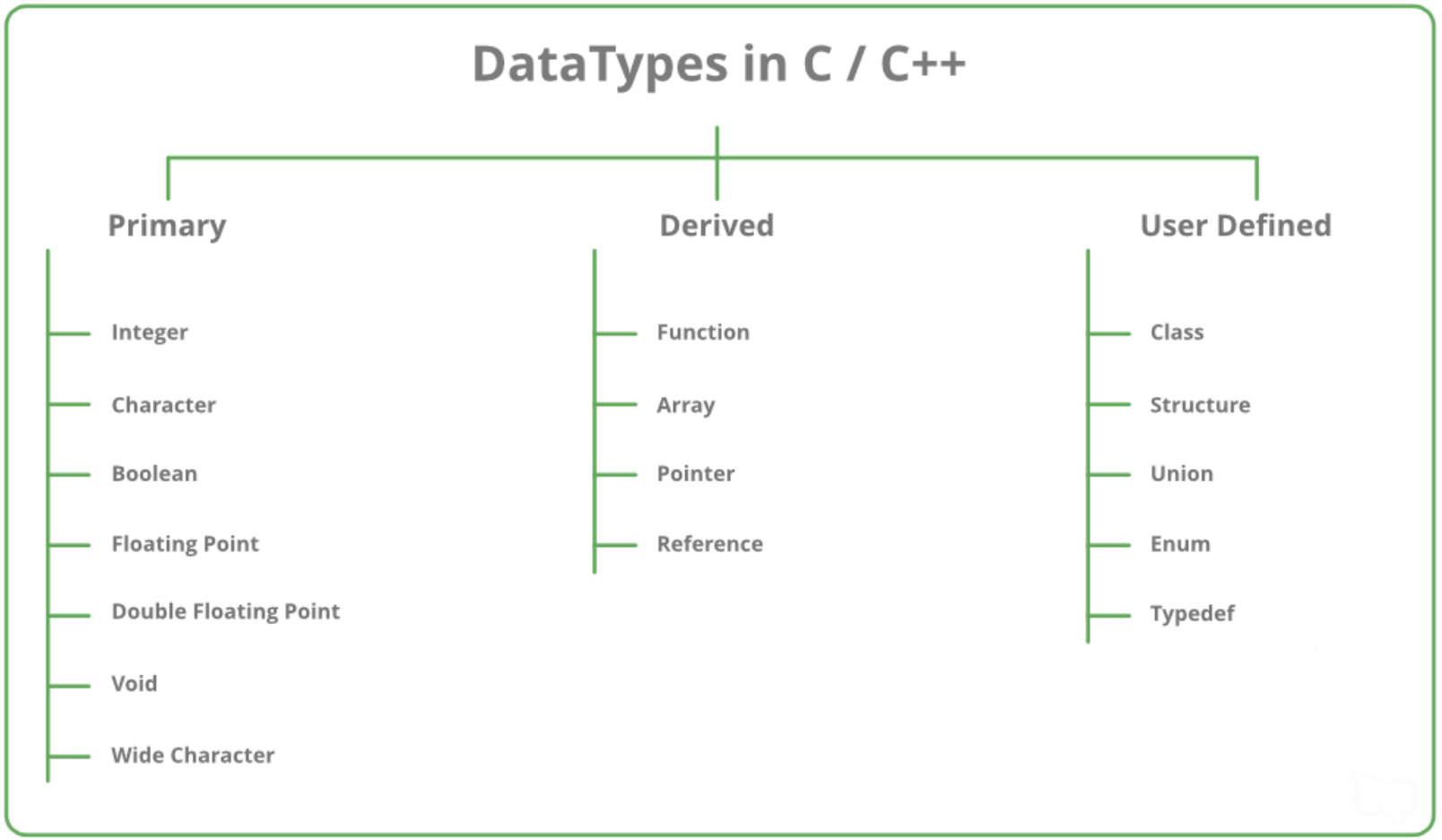
while statement

```
int i = 0;
while(i < 5){
   i++;
   std::cout << i << std::endl;
}
```

Data Types

Primitive Data Types: These data types are built-in or predefined data types and can be used directly by the user to declare variables. Example: int, char , float, bool etc.

Derived Data Types: The data-types that are derived from the primitive or built-in datatypes are referred to as Derived Data Types.



Functions

A function is a block of code or program-segment that is defined to perform a specific well-defined task. A function is generally defined to save the user from writing the same lines of code again and again for the same input. All the lines of code are put together inside a single function and this can be called anywhere required. main() is a default function that is defined in every program of C++.

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

// max here is a function derived type
int max(int x, int y)
{
    if (x > y)
        return x;
    else
        return y;
}
```

```
// main is the default function derived type
int main()
{
    int a = 10, b = 20;

    // Calling above function to
    // find max of 'a' and 'b'
    int m = max(a, b);

    cout << "m is " << m << endl;
    return 0;
}
```

Array

An array is a collection of items stored at continuous memory locations. The idea of array is to represent many instances in one variable.

```
#include <iostream>
using namespace std;
int main()
{
   // Array Derived Type
   int arr[5];
   arr[0] = 5;
   arr[2] = -10;

   // this is same as arr[1] = 2
   arr[3 / 2] = 2;

   arr[3] = arr[0];

   cout<<arr[0]<<" "<<arr[1]<<" "<<arr[2]<<" "<<arr[3] << endl;

   return 0;
}
```

Pointers

Pointers are symbolic representation of addresses. They enable programs to create and manipulate dynamic data structures.

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

int main()
{
    int var = 20;

    // Pointers Derived Type
    // declare pointer variable
    int* ptr;

    // note that data type of ptr
    // and var must be same
    ptr = &var;

    // assign the address of a variable
    // to a pointer
    cout << "Value at ptr = "
         << ptr << "\n";
    cout << "Value at var = "
         << var << "\n";
    cout << "Value at *ptr = "
         << *ptr << "\n";

    return 0;
}
```

Reference

When a variable is declared as reference, it becomes an alternative name for an existing variable. A variable can be declared as reference by putting '&' in the declaration.

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

int main()
{
    int x = 10;

    // Reference Derived Type
    // ref is a reference to x.
    int& ref = x;

    // Value of x is now changed to 20
    ref = 20;
    cout << "x = " << x << endl;

    // Value of x is now changed to 30
    x = 30;
    cout << "ref = " << ref << endl;

    return 0;
}
```

Classes and Objects

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

class Robot
{

   public:
   string name;
   int max_speed;
   int speed;

   void  increase_speed(int  speed_increase){
               speed = speed + speed_increase;
               if (speed > max_speed){
                   speed = max_speed;
               }
      cout << "The speed of " << name << " is " << speed << endl;
   }
};
```

```
int main(){
   Robot robot1;
   robot1.speed = 0;
   robot1.name = "robot1";
   robot1.max_speed=10;

   Robot robot2;
   robot2.speed = 0;
   robot2.name = "robot2";
   robot2.max_speed=20;

   robot1.increase_speed(20);
   robot2.increase_speed(20);

   return 0;

}
```

CMake File

An alternative to build your C++ project is to use a software tool known as CMake. CMake is one of the most popular tools for building C++ projects because it is relatively user-friendly (compared to the Linux makefile)

apt-get install cmake

Now, in the folder of your project, create a file CMakeLists.txt, with the following structure

```
cmake_minimum_required(VERSION 3.0)
set(CMAKE_BUILD_TYPE Release)
project(classes)
add_executable(
        classes
        classes.cpp
)
```

CMake File

Create then a new folder, inside the one of your project, called build, and move to that folder:

mkdir build
cd build

Once done, run

cmake..
make