University of Trieste

International School for Advanced Studies

The Abdus Salam International Centre for Theoretical Physics

Advanced Programming

Quick notes

Author:
Marco Sciorilli

Tuesday 25th May, 2021

Chapter 1

C++

1.1 Hello world

C++ is a statically typed language (i.e. the compiler must know at compile time the type of each symbol, and strongly typed language (i.e. each symbol has a unique type, and that type cannot change)

1.2 Type, Object, Value and Variable

1. Type: C++ is strongly typed and strongly checked language.It is a particular implementation of a concept.
2. Object: a portion of memory, a given amount of bytes.
3. Value: the sequence of bits contained in the memory occupied by the object. It is interpreted according to the type.
4. Variable: a named object.

1.3 Built-in types and simple ops

- Built-in types: int(4 bytes), doubles (8 bytes), char (1 bytes), booleans (1 bytes). TO MEMORIZE
- Custom types: everything else (e.g. strings).

Integers has a given amount of bytes, if we want to increase it, we have to use the keyword `long long`. A bit is used to store the sign. If we do not need it, we can use it with the keyword `unsigned`.
For double we can use `double` or `float` (32 bits, half the size of double, 64 bits).
Strings need the header #`include <string>`. Single quotes are for char, double quotes are for strings.
What is the difference between the equal sign and the universal and uniform initializer {}?

Universal and uniform initializer recognise the narrowing effect (initialize an integer with a double, a char as a string).
A cast force the reinterpretation of the value not according to its type, but to the type specified in parenthesis.
`const` is a declaration of intent: the value will not be changed. Use as much `const` as we can: for correctness of code(the compiler check if its true that the value does not change), and for better performance (i.e. more aggressive optimization). We cannot make a `constexpr` containing a variable: we cannot know its value at compile time.

1.4 Loops

`do {} while(condition);` is a while loop where the condition is checked at the end of the iteration.
`break` exit from the innermost loop.
`continue` skip an iteration.

1.5 Pointers, references, and functions

The portion of memory occupied by the object has an address (we know where the object is finished because each type has an associated length). We can know the address of a variable with the unary ampersand operator.
The reference create an alias to a variable (reference and the variable itself are interchangeable), their address is the same.
If instead we want to change the value of a variable through a pointer, we first have to dereference the pointer variable (*), i.e. "we enter the house the pointer is pointing to, once inside we can change the value".
A reference is a pointer which automatically dereference itself.
If a function does not return anything, we have to declare it as `void`.
Function declaration is different than function definition:

- Declaration: introducing a new symbol for the first time.
- Definition: defining the body of the function.

The compilation only require the declaration, the linker go to look for the definition of the function, to make the executable.

1.6 Watch out!

Not all floating point number can be correctly represented: most of them have some degree of error. So, to compare two floating number, we cannot use the normal operator. We to do instead, for example `std::abs(float-otherfloat) > 1e-15`: i.e. the difference is below a given threshold.

1.7 Keyword "auto"

With the keyword "auto" the compiler automatically deduce the type of a variable from it initialization value. With `auto` we should always initialize with the equal sign (otherwise the compiler can misinterpret the variable as another type).

1.8 Function overloading

We can have many function with the same name, but different number or types of argument. That is called function overloading.

1.9 Templates

To avoid the overload of equal functions which just have different variable types, we can use templates.
A template is a placeholder which will be feed by the compiler (it is up to the compiler to copy and paste the function, substituting the right types). It has the form `template<typename T>` where `T` is the placeholder for the type used in the function. We can specify the type to use when we call the function: e.g. `foo<int>(arguments)`.
What is specified in the template must be known at compile time, so if we specify a type (`<typename T, int N>`) than we have to specify the value when we call the function (e.g. `<int, 5>`); so variables can be used only if constant. The order is important. The specified order win over the deduced one.

1.10 Built-in arrays

Arrays in C++ do not have bound checking. When creating an array of built-in types, each cell is left uninitialized. In memory an array is a set of subsequent "cells" of memory.
An array decays to a pointer to its first element.
Built-in arrays live in the Stack (i.e. the portion of the RAM which is automatically erased when the function is over). Dynamic arrays on the other hand live in the Heap (free store, the memory is occupied until the user say otherwise). To create a dynamic array, we have to use the operator `new`. Stack is very small, the heap is the size of the RAM (minus Stack). To release the memory occupied by an array, we use `delete[] arrayname;`.

1.11 Resize built-in arrays

We cannot resize the array: we have to create a new bigger array, copy all the values from the first one.
A scope is defined by a simple open and close curly breaches (outside the scope, the symbols used inside of it are gone).

1.12 Const and pointers

We can specify const for a pointer itself (once we have specified that a pointer points to a variable, it cannot point to a different one). We can specify const for the variable the pointer is pointing to (when dereferencing the pointer, the variable cannot change, we can only read).

1.13 Special pointers

$**$ is the signature for a pointer to a pointer. So for example `char**` is a pointer to pointer to char. As an array decays to a pointer to its first element, and as a string is an array of char, `char** pcc` = `char* ppc []`.
`void*` is a pointer to void: can be initialized with the values of all the pointers we want, but before using it, we need to perform a cast toward a proper kind of pointer.
The difference between a reference and a pointer is that a reference **cannot** point to anything.

1.14 Matrices

Matrices are one long array. The number of columns must be stated.

1.15 `std::array` and `std::vector`

For loops can be done by `for (auto x :  a)`, where, if we want to modify `a` we have to use `auto`&.
`std::array` are as fast as built in arrays.
Command `b.at(number)` performs bound checking at run-time, with a small run-time penalty.
Instead of dynamic built-in array we can use instead `std::vector<type> name`{`elements`}, using curve braces is different: e.g. `(4,4)` means "vector of 4 elements, all with value 4".

1.16 Plain enum

`enum` is a new type, where we can introduce a new concept and name it. By default, integer values are assigned to the elements of `enum`. It is useful to make the code more readable.
`switch (variable)`{`case:  something; break;` } statement, where `case` go through the elements of `enum`, is used to implement different operations on different element of enum.

1.17 Scoped enum

It is implemented with `enum class`. In this case when we want to access an element of `enum`, we have to prepend `type::element` (like a class). Having to prepend the type solves the cases of ambiguity where different enum class has the same element.

1.18 Namespace

A namespace is something that enrich the name of a variable with the name of the namespace itself.
It takes the form `namespace choose_a_name { }`. Whatever is defined between the curly breaches, will have a name with the namespace prepended `choose_a_name::variable`.
Inside of the namespace, prependig the name of the namespace is not necessary.
`using namespace` extend to the global level all the symbols defined in the namespace.

1.19 Struct and class

We can introduce a new type using `class` or `struct`. They are the same, except for the fact that the default visibility of `struct` is public, while for `class` is private. They are a combination of variable and function.
The collection of all the public symbols is called the interface of the class (or struct). The distinction private and public is to guarantee to the user the the status of the object is always valid.
A pointer to a custom type follows the the same syntax as the one of a built-in type.
When we are dealing with an pointer to a custom type, members are accessed using the arrow ($->$) operator.

1.20 Operator overloading

With the keyword `friend` (and declearing the function) we can grant to our friend-functions or classes to the private data.

1.21 Constructors and destructor

How we can control how our custom type object is constructed? Using a constructor: they have the same name of the custom struct or class, do no return anything, and the argument are listed as for normal functions.
We can have constructor overloading (many constructors that just differs for the type or number of arguments).
The default constructor is called without any arguments. The destructor is $\sim$ `classname`.
To create a costructor outside of the class is just e.g. `Foo::Foo(arguments)`. After the closed parenthesis, it possible to initialize variables in curly braches after a :
e.g. `: _i{i},`
If not specified in the constructor, built-in types are left uninitialized, and for each of the custom type the default constructor is invoked.
Constructor are invoked from top to bottom, destructor in the other direction.

1.22 Template class

Combination of constructor and destructor is called RAII (Resource acquisition is initialization): is a design patter which allows to write clean code, because it tells us which resources we acquire, and to automatically delete it when we are done (we go out of scope)

1.23 Complier-generated ctors

`variable1 = std::move(variable2)` is as a swap: i.e. we take `variable2`, and we put it in `variable1`. The content of `variable2` has been stolen.
If we do not specify the them in the class (or struct), the compiler generate for us a lot of functions.
Aggregate initialization: if the type used does not have custom ctors, we can directly initialize the members.

1.24 Default and delete keywords

How to initialize default values for variable and functions? Declaring the value of a variable of a type is called **in-class initialization**. We can explicitly ask the compiler to generate (copy constructor, copy assignment, move constructor, move assignment) with the command `classname() = default;`.
We can ask the compiler to delete one of the default constructed aforementioned functions with command e.g. `classname(const classname&) = delete;`

1.25 Buggy vector

`std::unique_ptr<type[]> elem;` are pointers which express the idea of unique ownership. We can access the object a unique pointer is pointing to with the method `.get()`.

1.26 Copy and Move for a vector

Unique pointer is a smart pointer: they are as fast as raw pointers, but automatically implement RAII.There are two types of copies:

- Shallow copy: coping only the pointers. It is faster, but we have linked object at the end.
- Deep copy: create two separate objects, coping all the values stored in that portion of the memory.

Obviously unique pointers do not support copy semantic, but support move semantic.
Reference with only one & is called l-value reference. i.e. it has a name and can be on the left side of an assignment. Reference with two && is called r-value reference. i.e. it is something which has no name, and cannot be on the left side of an assignment.

1.27 Keyword "explicit"

A constructor that can be invoked with just one argument defines an implicit conversion between the argument of the constructor, to the type of the class itself.

If we want to avoid this implicit conversion, we have to mark the constructor as `explicit`.

1.28 exceptions

Error handling: `try{}`, with between curly breaches all the functions we want to try. If inside the try block an exception is thrown, we can check the type of exception has been thrown and test for test for different types in one or more `catch(type of exception)error message`. To categorize the kind of exceptions, we create a class to check the preconditions, which `throw` exceptions when needed.
That is not a good design strategy.
We can instead construct an error with the error message already inside.

1.29 assert

When the program is very long, it is useful that the error handling would be able to point at the line where the exception was thrown.
A macro is something defined with # `define`, is replaced by the compiler (are substitute).
Difference between error and assert: inserting checks slows down the code. We may want to turn this checks on and off depending on some compilation flag. Assert allows that.
By convention, the exceptions of the standard library print the message using the member function `.what()`. So also our library which check the exceptions also should implement the `what()`.
Assert are part of C, so they do not throw exception. They perform a simple check.

1.30 Stack unwinding

Stack unwinding is what happens when an exception is thrown. When an exception is thrown inside a `try{}` stack unwinding begins, so the destructor of all the objects constructed so far, is invoked.
We can have nested try and catch blocks. With the statement `throw;` we can re-throw inside a catch block the same exception with the same message. That allows to go backward until someone does not rethrow the exception. Eventually, we arrive at the main, where the problem is over.
`noexcept`: handling the exception is not free, in terms of computation time. If we are sure that a function cannot throw an exception, we have to mark it as `noexcept`, because the compile will produce a better/faster node.

1.31 Object composition

To implement the idea that a concept has a particular characteristic, we use object composition i.e. we define its characteristic inside the concept class.
At the opposite the idea that a concept is something, is implemented through the use of inheritance.

1.32 Inheritance and run-time polymorphism

Expressing inheritance e.g. `struct Dog :public Animal` {}. When building a constructor of a child class, we always have to put first the constructor of all the parents of that class.
Run-time polymorphism has no template, but has a reference to the base class.

1.33 Dynamic binding

Means that at run-time, we do not have to call the base class, but the functions of the derived class. It completes Run-time polymorphism. To do so, we have to mark the functions of the base class as `virtual` (as well as the destructor). In the derived class instead we `override` the function (it has to have the same name and same arguments). Pure virtual function are initialized to 0, and a class which has at least 1 pure virtual function becomes an abstract class. We cannot instantiate an object for another class.
When the parent class is templated, to access a member of the class, we have to use `this->member`.
Overloading does not work through inheritance. The solution is the syntax: name of the parent class:: name of the function e.g. `using foo::f;`.

Chapter 2

Python

2.1 How to use python

Two kind of way to use python: a non reusable one, inside the python shell (closing the shell, whatever was inside of it was gone); and a reusable way, writing the python code in a file.
So python is not a programming language, but rather a virtual machine (not an interpreter language): it reads the statement, converts them into bite code which is than compiled by a virtual machine, and eventually b the CPU.
`#!/usr/bin/env python3` automatically call the python3 when the file is executed.

2.2 Intro to jupyter notebook

jupyter notebooks have some built in magic commands.
Magic commands of the first type start with a single %, and run in one line. Magic commands of the first type start with two %%, are applied to all the rest of the cell
! execute a command on the underline shell.
Jupyter notebooks are usefull because they are resilient to errors: what has been done in a cell is not lost if an error occurs in another one.

2.3 Python surviving kit

All the types are particular implementation of objects. In python, integer numbers do not have a fixed amount of bits (they are allocated on the fly).For booleans, every number different from 0 is True. Floating point numbers do not have a specific precision.
With command `help()` we can access the documentation of an object.
`None` type is the only type which is not defined by a class.
Variables name in python are like post-it on box (where box are objects).
`id()` allows to check the identity of an object. Operator `is` allows to compare the id of two variables.
Indentation is done with 4 spaces, not a tab.

2.4 List

- Mutable
- Resizable
- Stored objects can change
- Objects can be of different type

slicing

`list[start:stop:step]`

2.5 Python Tuples

- Immutables
- Cannot add, remove, change objects once created.
- Slicing

We can create one with `tuple()`, or simply `object = a, b`.
We can unpack (i.e. extract objects from) a tuple in the same way:`a, b = object` , assigning _, to the element in the tuple we do not care about. Using the syntax $*$ put all the remaining objects of an interval inside a list. It can be obviously used only once in tuple unpacking. To unpack nested tuples, we have to repeat round parenthesis of the inner tuple.
Immutability: we are talking about the id of the object (its address). So for example we can change a list inside a tuple, because a list is referenced by an address to the first element.
IT IS BETTER TO NOT MIX MUTABLE AND IMMUTABLE OBJECTS.

2.6 Python dicts

- Unsorted set of pairs `key:value`
- Elements are accessed by `key` and not by offset
- Are mutable, so you can add, delete and change their `key:value` elements
- Highly optimized

Iterating on a dictionary only gives us the keys.
Method `.items()` returns a tuple on all the items in the dict.
`OrderedDict` is an alternative to dict which preserves the insertion order.

2.7 Set

Can be created using `set()`. In does not contain repeated values (like a mathematical set). Operations of union and intersection can be implemented with `union` and `intersection`.

2.8 Functions

We leave 1 empty line after a function, to make clear that it ended.
Positional argument: you pass only the element.
Keyword argument: you specify the argument of the element you are passing (i.e. `f(argument = element)`.
We can use $*$ before an argument, and the function will create a container, and fill it with all the argument passed (when you do not want to limit the number of elements passed as arguments). $**$ is used for keyword argument: it creates a dict.
Positional arguments comes before than key arguments.
Lambda functions are anonymous in line function which returns one values (so "return" is implied)
Functions are first-class object, aka:

- Created at runtime
- Assigned to a variable or element in a data structure
- Passed as an argument to a function
- Returned as the result of a function

If we want the user to use only keyword arguments, we can put an $*$ as the first argument in the function definition.

2.9 Modules and packages

A file whose name end in `.py`, and contains valid python code.
`__all__=[ ]` allows to specify the functions we want to import.
Many modules can be organized in packages.
`__init__.py` file can be empty, but must be present. If we want to import the whole package, we have to write `__all__=[ ]` listing the modules, in `__init__.py`.

2.10 Copy and deep copy

Shallow copy is the default. For example when coping a list, it do a deep copy only of the first level (the element of the list still points to the same memory space).
The function `copy()` once again only performs a deep copy only on the first level. If we want a true deep copy we have to use `deepcopy()`

2.11 Global and local variables

Python firstly read all the file, and decide which variables are global, and which are local. Defining a variable in a function after its use, even if the same variable was already declared outside of the scope of the function, generates an error, as python already labeled such a variable as local to the function (even if it has the same name of a local one. If we want to change a global variable inside the scope of a function, we have to flag the variable with the keyword `global`.

2.12 Nonlocal(free) variables

A variable which does not belong to the scope of a variable, but was part of the defining scope (e.g. the case of nested functions). By default mutable objects are free variables, immutable not. To make an immutable object a free variable, we have to flag it with the keyword `nonlocal`.
Closure: a function which retains the bindings of the free variables that exits when the function is defined, so that they can be used later when the function is invoked and the defining scope is no longer available.
In a function, never use a default empty container, use None instead.

2.13 Python Class

All the subscripted methods are called *dunder methods*. We can create one using e.g. `Class point(object):` where between parenthesis we express the inheritance.
`__init__` is the equivalent of the constructor. In python everything is public.
To make an overload of `print()` there is two way: `__repr__` and `__str__`. The first one has to be unambiguous i.e. if there are two different objects, than their representation has to be different (da riguardare).

2.14 Decorators

Can be implemented as a function whose argument is another function. It defines an inner function, and inside the inner function we add what we want to add to function passed as argument (the inner function is a closure).
To avoid losing documentation every time we use a decorator, we use `@functools.wraps`, another decorator, which decorates the inner function. To always pass the right amount of argument to the decorated function, we can use the $*$ and $**$ keywords for arguments.
Parametrised() decorators add a level, used with round parenthesis (to specify something on a lower level decorator).
Decorator can be implemented as function objects: we create a class with the name of the decorator. The constructor take as input self as well as the function to decorate. In this case to update the documentation we use `functools.update_wrapper(self, func)`.
Than we implement the dunder method `__call__`.

2.15 @property

We use it when we want to make an attribute of the class (a member variable) constant. Or to perform some specific action when an attribute is set.
We achieve the first goal by creating a methode decorated by @property which returns the attribute (set private in the constructor using the underscore).

2.16 Instance members

@classmetode (with cls in the place of self) used to modify only the class variables or create a new instance of the class.
@staticmethod do not modify instance or class variables.

2.17 Inheritance

To access the instance of a parent class, we use the keyword `super()`. The order of parenthood is determined by the order in which it is specified in the class parenthesis. To check the inheritance, we use command `isinstance` and `isclass`.
An abstract class is a class with at least one pure virtual method. An abstract method can be implemented in python with the decorator: `@abs.abstractmethod`, from the package `abc`, and the class inherit from `abs.ABC`.

2.18 Error handling

As in C++, but in python we `try` and `except`. Also we `raise` and exeption. But there is also the command `finally`, which is always executed.
`assert` is used to check is a condition is met. We can skip assert, to avoid slow down the code, with the flag `-O` at execution time. Never use parenthesis with `assert`.

2.19 Iterator and generator

While iterating through something in a function, `yield` return the data but remember the point reached by the iteration.
Generator something that gives you the next element on the fly. Doing a "tuple comprehension" return instead a *generator*. This operation is called generator expression.
Also, generator can be chained: when iterating through one, all the chained generator change.

2.20 Context manager

For dealing with files: `with open("filename", "w") as f:`, then whatever is printed will appear on the file. One the context manager is finished, the file is automatically closed. It achieve RAII pattern.
Context manager does not defines a scope, so variables defined inside of it, survives the closing of the end of it.
Context manager is implemented as a class with two methods: `__enter__` and `__exit__`.
`__enter__` has no arguments, while `__exit__` takes values that allow to check whether an exception was thrown.

2.21 Testing

When doing testing, we have to specify whether a function is explicitly written, or imported. We can check it using `if ' =='__main__'` . importing `unittest` allows us to test every single function. We have to write a class which inherit from `unittest.TestCase`, define functions which begin with `test_`, and internally use wrappers around assert. At the end, to run the test, we put a

```
if ' =='__main__':
import doctest
doctest.testmod()
unittest.main()
```

Where `doctest` allows to keep the documentation consistent.
`pytest`, converts all the function which begins with "test" into test.