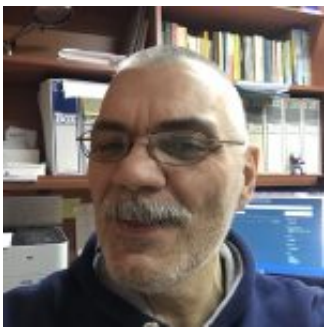
C++ bunch 1



(C) Prepared by Panaos Zafeiropoulos

Start learning C++ classes

This is a bunch of tiny C++ projects for iroductory/learning purposes on C++ classes. **makefile** is included in each one of the projects.

Find the souces here

class1

-A very simple example of a basic class declaration and definition. -Declaration is separated and takes place in a header file, and Definition in a source code file.

class1a

An updated version of the **class1** project.

Updates only in the main class: main.cpp

- We use command-line parameters/arguments as input values for our variables (the 2 integer variables for calculation).
- We also use another "free" function (available only in main.cpp) for outputting the passed-in arguments.

- We use only the first 2 parameters as input values for x and y variables.

---

class1b

An updated version of the **class1** project.\ However, this is actually a sequel, based on the previous project: **class1a**

Updates only in the main class: main.cpp

- We added user interactive input for input variables (comma separated integers), if no command-line parameters/arguments passed-in with the executable execution.

---

class1c - "Passing by Reference" function

An updated version of the **class1** project. However, this is actually a sequel, based on the previous project: **class1b**

Updates only in the main class: main.cpp

- We added a "free" function, named "getintegers". It deals with the interaction with the user for providing (or not) the 2 (comma separated) integer values. Thus we avoid having some boilerplate code in the main().
- The function uses arguments passed in "**by reference**" (C++). The **&** (address of) operator denotes values passed by "pass-by-reference" in a function.
- Passing arguments "by reference" in a function, means that inside the function we can change the values of those parameters. The function does not preserve their values. So, the changed values are reflected into the calling scope (e.g. in the main() function in our case). Thus the "getintegers" function can return nothing. It can be just a void function.
- One more note is that in pure C there is no such "pass by reference" functionality. In C, **&** means "address of" and is a way to formulate a pointer from a variable.

---

class1d - "Passing by Const Reference" function - return an integer array (actually

a pointer to an integer array)

An updated version of the **class1** project. However, this is actually a sequel, based on the previous project: **class1c**

We added another free function "getintegersarr" which is an alternative to the previous "getintegers", dealing with the interaction with the user for providing (or not) the 2 (comma separated) integer values. This time we use "pass by **const reference**", and the "getintegersarr" returns an array of the 2 integers.

Overloading could have been a good option, however the previous "getinteger" function cannot be overloaded, because it was a void function and moreover, its signature/prototype remains the same, even we use the const modifiers. So, we use the new "getintegersarr" function.

Updates only in the main class: main.cpp

- The "getintegersarr" function uses arguments passed in "**by const reference**" (C++). Generally, passing an argument by reference, is considered more efficient. However, when we want to ensure that the arguments passed in will not be changed, then we can pass them "by const reference". The **const** modifier ensures the value of an argument will be remained intact.4

Some notes about the "getintegersarr" function

- We want to use "pass by reference" for efficiency
- We want the parameters passed-in to remain unchanged. Thus, we use the **const** modifier.

- We want to return an array. However -generally-, we cannot return an array from a function.

- A first commonly used approach is to return a pointer, pointing to the integer array, that the function returns.

- Here we use a local **static** variable for the array to be returned.

- There are also other approaches for returning an array - actually a pointer to array, e.g. by using dynamically allocated array: int* arr = new int[2]; however after calling the function and getting the returned array, we have to clear it, e.g. by using the delete or free(), in order to avoid memory leaks.

- Using a local static array is a preferable approach. This is because the lifetime of a

static variable is throughout the scope it resides, and here the scope is within the function. Thus there is no need to delete or free the array and its memory allocated.

—

class1e - "Passing by Const Reference" function - Using a struct wrapper to return an integer array

An updated version of the class1 project. However, this is actually a sequel, based on the previous project: class1d

Here, we use a very very simple struct, as a wrapper of an integer array of (fixed) size 2. This is the array actually holding the user input. In the main, we use a new free function of type of the struct, that it returns the array of the user input, wrapped in the struct.

Short intro to C++ structures - struct

In this version we use a **struct**. **structs** in C++ are classes that by default use public access modifier for all of their members (variables, functions, etc). A **struct** is a user-defined data type that combines logically related data items of different data types like float, char, int, etc., together. Moreover, it cannot have null values. A commonly used approach ****is to use structs as **POD**s (C **Plain-Old-Data** structures). A struct with no modifiers or methods is a C POD struct. This gives C++ a backwards compatible interface with C libraries. *"A POD-struct is an aggregate class that has no non-static data members of type non-POD-struct, non-POD-union (or array of such types) or reference, and has no user-defined copy assignment operator and no user-defined destructor."* A struct instance is considered as a “struct variable”

class2 - class constructor with member initialiser list

A second example based on class1 project.

In this example we use a class **custom constructor** **with parameters** in the header. The constructor initialisation uses member initialiser list. A **member initialiser list** starts with a

colon, followed by member names and their initialisers, where each initialisation expression is separated by a comma.

In our example, this is how we declare an initialisation list:

```
MathUtils(int aa, int bb) : a{aa}, b{bb}{};
```

This is the preferred way of initialising class data members. -No need to do anything else in definition code. -We can invoke the class like that:

```
MathUtils m_utils(99, 32);
```

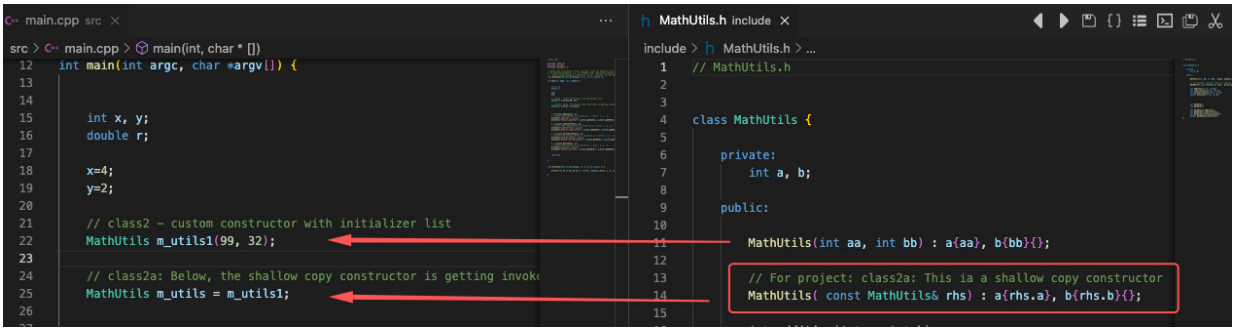
---

class2a - Shallow copy

An update (first update) based on the previous **class2** project.

In the previous example (**class2**) we used a (public) class **custom constructor with parameters**. The constructor initialisation used a **member initialiser list**. (We declared/defined it in the header).

In this example we add a **shallow copy** constructor.



The copy constructor has a special parameter signature:

```
MyClass(const MyClass& rhs)
```

In a case where we use a custom constructor with parameters, the shallow copy constructor looks like:

```
MyClass(const MyClass& rhs): x{ rhs.x }, y{ rhs.y }
```

---

class2b - Deep copy

A second update based on the previous **class2** project.

In the previous example (**class2a**) we used a (public) class **custom constructor with parameters**, and a Shallow copy constructor.

Here we are going to use a pointer member parameter - int * p

Then, the 'regular' constructor initialisation using a **member initialiser list** is similar to:

```
MyClass(int xx, int pp) : x{ xx }, p{ new int{pp} } {}
```

The **deep copy** constructor has a signature like the one below:

```
MyClass(const MyClass& rhs): x{ rhs.x }, p{ new int {*rhs.p} }
```

What actually have to do is to use the **new** keyword, to allocate a new memory for the pointer member parameter

---