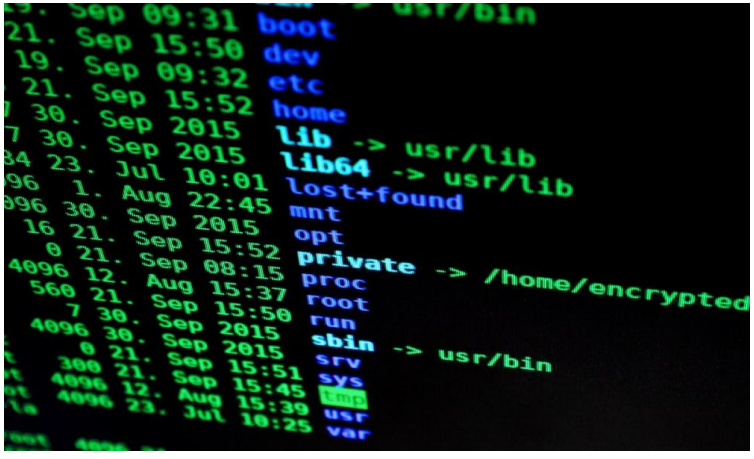
Specialized Numerical Methods for Transport Phenomena

Advanced C++ Programming

Bruno Blais and Laura Prieto Saavedra

Department of Chemical Engineering
Polytechnique Montréal

January 16, 2025



Outline

Function overloading

Function templates

Function parameters

Objects

STL Library

Iterators

Conclusion

Outline

Function overloading

Function templates

Function parameters

Objects

STL Library

Iterators

Conclusion

Common scenario

We need a function that calculates the sum of two integers:

```
int add_numbers(int a, int b)
{
    return a + b;
}
```

This works fine... but what if we want to have a function that also adds two doubles?

```
double add_numbers_double(double a, double b)
{
    return a + b;
}
```

Common scenario

We need a function that calculates the sum of two integers:

```
int add_numbers(int a, int b)
{
    return a + b;
}
```

This works fine... but what if we want to have a function that also adds two doubles?

```
double add_numbers_double(double a, double b)
{
    return a + b;
}
```

Basically the same function with different name and parameter types. In a big code this quickly becomes hard to handle!

What is the solution?

In C++, function overloading allows us to create multiple functions with the same name with different parameter types:

```
int add_numbers(int a, int b)
{
    return a + b;
}

double add_numbers(double a, double b)
{
    return a + b;
}
```

The compiler will be able to differentiate these functions as follows:

```
add_numbers(1, 2); // will call 1st version
add_numbers(2.1, 4.3); // will call 2nd version
```

How does the compiler differentiate?

- Based on types of parameters (as in the previous example)
- Based on the number of parameters:

```
int add_numbers(int a, int b)
{
    return a + b;
}
int add_numbers(int a, int b, int c)
{
    return a + b + c;
}
```

Note: the return type of the function is not considered when differentiating overloaded functions.

Outline

Function overloading

Function templates

Function parameters

Objects

STL Library

Iterators

Conclusion

When do we need this?

We can use function overloading for the following example:

```
int add_numbers(int a, int b)
{
    return a + b;
}

double add_numbers(double a, double b)
{
    return a + b;
}
```

However, note that the implementation is exactly the same for both versions! And in fact, it could be used also for other types, e.g., long or long double. This is again hard to maintain and a source of common errors in big codes.

When do we need this?

We can use function overloading for the following example:

```
int add_numbers(int a, int b)
{
    return a + b;
}

double add_numbers(double a, double b)
{
    return a + b;
}
```

However, note that the implementation is exactly the same for both versions! And in fact, it could be used also for other types, e.g., long or long double. This is again hard to maintain and a source of common errors in big codes. Can we write a single version of `add_numbers` that works with any type?

What is a template?

In C++ templates were created to simplify the process of working with different data types. We can simply create a single *template* of a function, using a "placeholder" type, also known as, template type `T`:

```
template <typename T> // template parameter declaration
T add_numbers(T a, T b)
{
    return a + b;
}
```

Once the template is defined, the compiler can use it to generate as many overloaded functions as needed. The result is the same as a bunch of overloaded functions but easier to create and mantain.

Usage

In C++

Templates are used extensively in many C++ containers (vector, lists, maps, etc.)

In `deal.II`

Templates will be used to parametrize the dimensionality of the problem, among many other things!

Outline

Function overloading

Function templates

Function parameters

Objects

STL Library

Iterators

Conclusion

What is a parameter?

In the function declaration we have *parameters*, e.g., a and b:

```
int add_numbers(int a, int b)
{
    return a + b;
}
```

On the other hand, when we call the function we have *arguments* that are being passed to the function:

```
int x = 2, y = 3;
add_numbers(x, y);
```

What is a parameter?

In the function declaration we have *parameters*, e.g., a and b:

```
int add_numbers(int a, int b)
{
    return a + b;
}
```

On the other hand, when we call the function we have *arguments* that are being passed to the function:

```
int x = 2, y = 3;
add_numbers(x, y);
```

How does this work? Through a process called **passing by value** where all the parameters are created as variables, and the value of each argument is copied into the matching parameter $\rightarrow$ copying makes things slow for complicated types...

Passing by reference

We can do better by using pointers and their addresses, since this avoids copying the argument:

```
int add_numbers(int& a, int& b)
{
    return a + b;
}
```

If we call the function:

```
int x = 2, y = 3;
add_numbers(x, y);
```

Now, when the function uses a and b, it is in fact accessing the actual arguments x and y. This has no cost and no copy of the arguments needs to be made.

Note: this means that if you change a and b inside the function, now the changes will affect the actual variables x and y. Unless you use `const`.

Outline

Function overloading

Function templates

Function parameters

Objects

STL Library

Iterators

Conclusion

Object-Oriented Programming

Until now we have seen how to define variables and functions.
If you look objects around you, e..g., books, buildings, etc, there are two major components:

- Attributes: weight, color, size, etc.
- Behaviors: open, close, etc.

Object-Oriented Programming

Until now we have seen how to define variables and functions.
If you look objects around you, e..g., books, buildings, etc, there are two major components:

- Attributes: weight, color, size, etc.
- Behaviors: open, close, etc.

The idea of OOP is to create program-defined data types that comprise both attributes (variables) and behaviors (functions). These types are called objects and a program can have one or several of them, and they can communicate with each other by sending messages.
The main benefits of OOP are:

- It allows for the creation of reusable code.
- It promotes code organization and modularity.
- It makes it easier to maintain and modify code over time.

How to define a class?

```
class Student
{
public:
    Student(int ID)// Constructor
    {
        id = ID;
        std::cout << "Student being constructed" << std::endl;
    }
    // Member functions
    void printID()
    {
        std::cout << id << std::endl;
    }
private:
    // Member variables
    int id;
};
```

We could add more variables such as age or program of study, and we could add more functions to print the different information.

How to use these objects?

In the main function, we can then create different students as follows:

```
int main()
{
    Student student_1(11030);
    student_1.printID();

    Student student_2(11031);
    student_2.printID();

    return 0;
}
```

At runtime the object is instantiated, the memory is allocated, and the constructor is called.

Templated classes

For example, we can have a `Number` class:

```
template <class T>
class Number
{
public:
    Number(T n) : num(n) {} // constructor

    T getNum() {
        return num;
    }
private:
    T num;
};
```

Then we can create objects as follows:

```
Number<int> numberInt(7);
Number<double> numberDouble(7.7);
```

Why OOP for scientific software?

Many things we will use in scientific computing are well represented by objects. Notably, when using the finite element method:

- A mesh
- Degrees of freedom
- Quadratures
- Matrices
- Linear solvers
- and so on and so forth...

Don't be scared...

You do not need to master OOP to succeed in this class. Objects will be introduced step by step. You will see that they will actually make our life significantly easier when we start manipulating complex concept.

Don't be scared...

You do not need to master OOP to succeed in this class. Objects will be introduced step by step. You will see that they will actually make our life significantly easier when we start manipulating complex concept.

Ask questions whenever you have hesitations. We know this stuff, we will be glad to explain everything you need. Remember, we will also follow your pace.

Outline

Function overloading

Function templates

Function parameters

Objects

STL Library

Iterators

Conclusion

The Standard Library

When you code you probably notice that we reuse the same concepts over and over again: loops, arrays, strings, etc. Therefore, C++ comes with a library that is full of reusable classes.
It contains a collection of classes that provide templated containers, algorithms and iterators.
Advantages:

- You do not have to write everything from scratch.
- Reusing classes allows you to avoid errors and long hours of debugging.
- These are well documented classes.

Disadvantages:

- The library is big and complex, you need to have a basic idea of templates.

The only way of getting used to the library is by using it!

STL containers

There are three categories:

- Sequence containers: maintain the ordering of the elements and you can choose where to insert your element by position, e.g., `std::array`, `std::vector`, `std::list`.
- Associative containers: automatically sort their inputs, e.g., `std::set`, `std::map`.
- Container adapters: adapted to specific uses, e.g., `std::stack`, `queue`.

STL iterators

Objects that can iterate over a container class without having to know how the container is implemented. They are better visualized as a pointer to a given element in the container. Each container will have four basic functions:

- `begin()`: returns an iterator representing the beginning of the elements in the container
- `end()`: returns an iterator representing the element just past the end of the elements
- `cbegin()`: const (read-only) iterator
- `cend()`: const (read-only) iterator

STL algorithms

Generic algorithms for working with the elements of the container classes, for example, they allow you to:

- Search
- Sort
- Insert
- Reorder
- Remove
- Copy elements

Note: they are implemented using iterators and they are supposed to work with all containers.

Outline

Function overloading

Function templates

Function parameters

Objects

STL Library

Iterators

Conclusion

Why do we need them?

Iterating over an array (or any other structure) of data is very useful and common when programming. We could use loops and an index (for and while loops), however:

- this is not concise
- it works only if the container (e.g., the array) provides direct accessing to elements

Another way of iterating through a container is to use **range-based for-loops**, which work for different structures:

- Arrays
- Lists
- Trees
- Maps
- ...

How does it work?

An **iterator** is an object designed to traverse through a container. The simplest kind of iterator is a pointer, which works for data stored sequentially.

```
#include <array>
#include <iostream>
int main()
{
    std::array new_array<int,3> {1, 2, 3};
    int* begin = &new_array[0];
    int* end = begin + new_array.size();
    std::cout << "new_array contains:";
    ffor (int* it = begin; it != end; ++it)
      std::cout << " " << *it;
    std::cout << std::endl;
    return 0;
}
```

If the pointers are already defined...

For example `std::array` provides the functions `begin()` and `end()` and they can be used as follows:

```
#include <array>
#include <iostream>
int main()
{
    std::array new_array<int,3> {1, 2, 3};
    std::cout << "new_array contains:";
    for (auto it = new_array.begin(); it != new_array.end();
        ++it )
      std::cout << " " << *it;
    std::cout << std::endl;

    return 0;
}
```

It can be even simpler...

```
#include <array>
#include <iostream>
int main()
{
    std::array<int,3> new_array{ 1, 2, 3 };
    std::cout << "new_array contains:";
    for (int i : array)
      std::cout << " " << i;
    std::cout << std::endl;

    return 0;
}
```

Behind the scenes, the range-based for-loop calls begin() and end(). Iterators are broadly use in deal.II, therefore, it is important for you to know this concept.

Outline

Function overloading

Function templates

Function parameters

Objects

STL Library

Iterators

Conclusion

In Conclusion

C++

You have now learned all the basics and some advanced concepts on C++.

Our usage of it will be minimal

You need to be able to understand these concepts to use the `deal.II` library. However, most of the "hard work" is hidden for us. You will get used to the syntax with practice and time. After all, the only way of learning how to code is to do it yourself! Of course, while having fun with it!