More C++ syntax

Raul P. Pelaez

October 31, 2024

Contents

1 Functions

We cannot keep all the code in the main function. We should order our code into compact, reusable and independent blocks of code. Most basic form of this is a function. A function is a block of code that has a name and can be called from other parts of the code. Functions can have parameters and return values.

The main syntax of a function, in pseudocode, is:

Source code

```
[return type] [function_name]([type1] arg1, [type2] arg2, ...) {
  // sequence of statements
  return value;
}
```

Example

Source code

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

// This is a function declaration
// Similarly to variables, we can define and declare functions at the same time
// so for this example the delaration is not strictly needed.
// int add(int a, int b);

// This is the definition of the function
int add(int a, int b) {
  //Functions consist of a sequence of statements
  // and can return a value with a type specified in the function signature
  auto result = a+b;
  return result;
}

int main() {
  int x = 5;
  int y = 3;
  int z = add(x, y);
  cout << z << endl;
  return 0;
}
```

The `auto` keyword plays a big role in functions. We can leverage it to add some generality to our functions. For instance, the function above works only for integers, but we can make it work for any type that supports the "+" operator by using auto:

Source code

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

auto add(auto a, auto b) {
  auto result = a+b;
  return result;
}

int main() {
  int x = 5;
  int y = 3;
  int z = add(x, y);
  cout << z << endl;
  double d = 3.14;
  double e = 2.71;
  double f = add(d, e);
  cout << f << endl;
  return 0;
}
```

There is a much more powerful mechanism in C++ to write generic code, called "templates". We will see them in the future.
While you should learn to use `auto` in this way, keep in mind that it can make the code harder to read and obtuse. More importantly, there are some limitations to using `auto` in this way. For instance when separating the declaration and definition of a function and splitting compilation between different files.

Advanced

`auto` as a return type is a C++11 feature, although back then it required specifying the type as a "trailing return type", like this:

Source code

```
auto add(auto a, auto b) -> decltype(a+b) {
  auto result = a+b;
  return result;
}
```

C++14 eliminated this requirement.
On the other hand, using auto for the argument types is a C++20 feature. This is a very recent addition to the language, and it is not yet supported by all compilers. Additionally, the reason why this works is quite subtle and has to do with "Concepts", a C++20 language feature that extends the template system.

Sometimes, the function needs to keep track of some state between calls. This can be done by using static variables. These variables are initialized only once, and keep their value between calls. For instance, we can write a function that counts how many times it has been called:

Source code

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

int count_calls() {
// This variable will keep its value between calls.
// It will be initialized only once, the first time the function is called.
  static int count = 0;
  count++;
  return count;
}

int main() {
  cout << count_calls() << endl;
  cout << count_calls() << endl;
  cout << count_calls() << endl;
  return 0;
}
```

Advanced

Lambdas: Anonymous functions

Sometimes we need a function that is used only once, and we don't want to bother giving it a name. Or maybe the function we have to define does not have a clear use outside the current scope, so we do not want to pollute the namespace with it.

This is where lambdas come in. Lambdas are anonymous functions that can be defined in place. They are very useful when we need to pass a function as an argument to another function, like in the following example:

Source code

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main() {
  vector<int> v = {1, 2, 3, 4, 5};
  // We want to count how many elements are greater than 3
  auto condition_function = [](int x) { return x > 3; };
  int count = count_if(v.begin(), v.end(), condition_function);
  cout << count << endl;
  return 0;
}
```

The body of a lambda does not have access to the scope above it by default, but we can "capture" some or all variables from the scope by using the square brackets. For instance, we can modify the previous example to count how many elements are greater than a variable "threshold":

Source code

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
  using namespace std;

int main() {
  vector<int> v = {1, 2, 3, 4, 5};
  int threshold = 3;
  auto condition_function = [threshold](int x) { return x > threshold; };
  int count = count_if(v.begin(), v.end(), condition_function);
  cout << count << endl;
  return 0;
}
```

We could had also used "&" to capture all variables by reference, or "=" to capture all variables by value.

There is a lot more to lambdas, but this is a good starting point.

In C++ we have the concept of "function overloading". This means that we can define multiple functions with the same name, as long as they have different parameters. The compiler will choose the correct function to call based on the parameters we pass. For instance, we can define a function that needs two arguments, and another that needs three:

Source code

```cpp
#include <iostream>
using namespace std;
void print_with_prefix(string message, string prefix) {
  cout << prefix << message << endl;
}
void print_with_prefix(string message) {
  cout << "Default:" << message << endl;
}
int main() {
  print_with_prefix("Hello", "Prefix: ");
  print_with_prefix("World");
  return 0;
}
```

In this particular case, we could have solved the problem by using another feature: default arguments. We can define a default value for a parameter in the function signature. This way, we can call the function with fewer arguments, and the default value will be used for the missing ones:

Source code

```cpp
#include <iostream>
using namespace std;
void print_with_prefix(string message, string prefix = "Default: ") {
  cout << prefix << message << endl;
}
int main() {
  print_with_prefix("Hello", "Prefix: ");
  print_with_prefix("World");
  return 0;
}
```

1.1 Key new concepts

Info

Functions
Functions are blocks of code that can be called from other parts of the code. They can have parameters and return values. Functions can be declared and defined separately, or at the same time. Functions can be made generic by using the `auto` keyword.

Info

Static variables
Static variables are variables that keep their value between calls to a function. They are initialized only once, the first time the function is called. You can define a static variable inside a function by using the `static` keyword as an attribute for a variable.

Info

Function overloading
Function overloading is the ability to define multiple functions with the same name, as long as they have different parameters. The compiler will choose the correct function to call based on the parameters we pass.

Info

Default arguments
Default arguments are values that are used when a function is called with fewer arguments than expected. You can define default arguments in the function signature.

2 Custom types

In C++, we can define our own types. This is done with the keyword "struct" or "class".
Contrary to C structs, in addition to data C++ structs can have functions (called methods) and access control (public, private, protected). The only difference between a struct and a class is that the members of a struct are public by default, while the members of a class are private. For instance, we can define a struct to represent a point in 2D space:

Source code

```
// Notice we do not need typedef here
struct Point {
  double x;
  double y;
  void print() {
    cout << "(" << x << ", " << y << ")" << endl;
  }
};
int main() {
  Point p;
  p.x = 1.0;
  p.y = 2.0;
  p.print();
  return 0;
}
```

If we want to use the class keyword, we can define the same struct as a class:

Source code

```
class Point {
// We need to mark the members as public in order to access them
public:
  double x;
  double y;
  void print() {
    cout << "(" << x << ", " << y << ")" << endl;
  }
};
int main() {
  Point p;
  p.x = 1.0;
  p.y = 2.0;
  p.print();
  return 0;
}
```

Suggestion

Try to remove the "public:" keyword from the class definition and see what happens when you compile the code. It is important that you get familiar with the compiler errors.

Advice

You already encountered a class in C++, the `std::string` class. When you called its `size()` method, you were calling a method of the class defined in a similar way as the above code.

Suggestion

Use your editors capabilities to travel to the definition of the `std::string` class. Standard library code is not the easiest to read, but it is a good exercise to try to understand it.

Advice

By convention, classes usually are named with a capital letter at the beginning.
The difference between class and struct is almost a cosmetic one. The choice between them is mostly a matter of style. Some people prefer to use structs for simple data containers, and classes for more complex objects. Others use classes for everything.

Classes in C++ are a foundational and immensely in-depth concept. We will learn about it in detail in the future.

C++ also has type aliasing, which consists in giving a type a second name. This is done with the keyword "using". For instance, we can define a type alias for a vector of integers:

Source code

```cpp
#include <iostream>
#include <vector>
using namespace std;
using IntVector = vector<int>;
int main() {
  IntVector v = {1, 2, 3, 4, 5};
  for (auto x : v) {
    cout << x << " ";
  }
  cout << endl;
  return 0;
}
```

From now on the keyword "typedef" for type aliasing is forbidden.

2.1 Key new concepts

Info

Structs and classes
Structs and classes are used to define custom types in C++. They can have data members and functions (methods). Methods and members can have access control (public, private, protected). The only difference between a struct and a class is that the members of a struct are public by default, while the members of a class are private.

Info

Type aliasing
Type aliasing allows us to refer to a type with a different name. This is done with the keyword `using`.
This is the only use of the `using` keyword besides when it appears in the `using namespace` construction.

3 The standard library

C++ has a powerful standard library with a lot of functionality. We have already seen some of it, like the `std::string` class, or the `std::vector` class. We also explored some of the algorithms in the `<algorithm>` header, like `std::count_if` or `std::accumulate`.

You should keep this page (`https://en.cppreference.com/w/cpp/standard_library`)around as a reference, it will give you an eagles view of the standard library. To name a few, we have utilities to handle time:

Source code

```cpp
#include <iostream>
#include <chrono>
using namespace std;
int main() {
  auto start = chrono::high_resolution_clock::now();
  // Do something
  auto end = chrono::high_resolution_clock::now();
  chrono::duration<double> elapsed = end - start;
  cout << "Elapsed time: " << elapsed.count() << "s" << endl;
  return 0;
}
```

Many algorithms to deal with data, like sorting, summing, finding elements, etc:

Source code

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int main() {
  vector<int> v = {5, 3, 1, 4, 2};
  sort(v.begin(), v.end());
  for (auto x : v) {
    cout << x << " ";
  }
  auto max = *max_element(v.begin(), v.end());
  cout << endl << "Max: " << max << endl;
  auto sum = accumulate(v.begin(), v.end(), 0);
  cout << "Sum: " << sum << endl;
  // Find the first element greater than 3. This function returns an iterator
  ↪  to first element that satisfies the condition
  auto element = *find_if(v.begin(), v.end(), [](int x) { return x > 3; });
  return 0;
}
```

It also provides containers, like sets, maps, queues, etc:

Source code

```cpp
#include <iostream>
#include <set>
#include <map>
#include <queue>
using namespace std;
int main() {
  set<int> s = {1, 2, 3, 4, 5};
  map<string, int> m = {{"one", 1}, {"two", 2}, {"three", 3}};
  queue<int> q;
  q.push(1);
  q.push(2);
  q.push(3);
  while (!q.empty()) {
    cout << q.front() << " ";
    q.pop();
  }
  cout << endl;
  return 0;
}
```

Also input/output utilities, like file handling, string manipulation. Random numbers, multi-threading, general utilities (like tuples), memory handling...

Advice

Whenever you need to solve a problem in C++, check the standard library first. It is very likely that there is a function that does what you need or can help you get there faster.

4 Exercises

We will be going over some exercises with increasing complexity. Exercises have a main goal with a set of milestones to get there. Please keep a record of your progress by saving a different file for each milestone. You can call them calculator1.cpp, calculator2.cpp, etc.
Some milestones are advanced and optional. They are there to challenge you and to help you learn more about the language.

4.1 Calculator

Goal

Write a program that will take two numbers and an operation and will return the result of the operation.
Example

Source code

```
$ ./calculator
Enter your operation:
2 + 3[Enter]
5
```

[Enter] means that you should press the Enter key. The line "2 + 3" is the input to the program the user writes, and the line "5" is the output of the program.

Milestone

Write a program that prints the initial message and reads the input from the user.

Hint

- Use "std::cout" and "std::cin" to print and read from the console.
- Take into account that the user will write the operation in the form "number operator number".
- The operator will be interpreted by cin as a string.

Advanced milestone

If needed, modify your program so that both these inputs are valid: "2 + 3" and "2+3".

Hint

- The character "2" will be interpreted as a number, but sequence "+3" will be interpreted as a string when there are no spaces.

Milestone

Write the logic to parse the input and return the result depending on the operator.
Write two versions of this program, one using if-else statements and another using a switch statement.
In both cases, print an error message if the operator is not recognized, i.e. an invalid input such as "2 & 3".

Advanced milestone

If you input floating numbers with many digits, like "1.23456789 + 2.3456789", you will see that the printed result is not accurate. Modify your program so that it prints the result with a fixed number of decimal places.

Hint

- We do not want to use printf, but luckily the standard iomanip library can help us.

Advanced milestone

Make the program more robust by checking that the input is correct. If the input is not correct, print an error message and ask the user to input the operation again.
For instance, your program should fail if the user inputs "2 + + 3" or "2+-++3".

Hint

- Note that the inputs "2+-3" or "-2+3" are valid.
- `std::cin>>something` is an expression which will return a boolean value. cin is a kind of `basic_istream` object, check the `operator!` in the documentation for basic_istream.

Milestone

Write a function that takes two numbers and an operator and returns the result and reorder your code accordingly.

Advanced milestone

Instead of a long list of if-else or switch statements, use a map to store the operations and their corresponding functions.

Hint

- You can use a map from the standard library, it lives under the <map> header.
- The map should have a string as a key and an `std::function` (<functional> header) as a value.
- Use lambda functions to define and register the operations. For example, you can define the lambda function for the addition operation as `[](double a, double b){return a+b;}`.

4.2 Vector accumulation

Goal

Write a program that will generate a random vector of numbers and will compute the sum of all the elements.
In C++, random numbers are generated using a combination of three elements:

- A random device that will provide a seed.
- A random number generator engine.
- A distribution that will define the range of the random numbers.

You can generate a random number like this:

Source code

```
#include <random>
#include <iostream>
using namespace std;
int main(){
  random_device rd;
  auto seed = rd();
  mt19937 gen(seed);
  uniform_int_distribution<int> dis(1, 6);
  cout<<dis(gen)<<endl;
  // Every time you call dis(gen) you will get a different random number
  ↪  between 1 and 6.
  cout<<dis(gen)<<endl;
}
```

There are many distributions to choose from. Note that the example distribution is for integers, you can choose one for floating numbers as well.
As well as many engines.

Milestone

Write a program that fills a vector with random numbers and prints them. For the moment use integer numbers.

Hint

- You can use the `push_back` method of the vector to add elements.

Milestone

Write a function that takes a vector of numbers and returns the sum of all the elements.

Advanced milestone

Write another version of the function that does not use a loop.

Hint

- Use recursion (a function that calls itself).

Milestone

Write a test for the function by comparing the result of your function with the result of the `std::accumulate` function.

Hint

- The header `numeric` contains the `std::accumulate` function.

Advanced milestone

Make sure your program works for floating numbers, try it out with a large vector (e.g. 1e6 elements).
Some comparisons will probably fail, can you fix it?

Hint

- Look at the documentation for std::accumulate.
- Learn about Kahan summation.

4.3 Histogram

Goal

Write a program that will generate a vector with uniform random numbers in a given range and will print a histogram of the numbers.
The histogram will be represented by a series of bars, each bar representing the number of elements in a certain range. Normalize the histogram so that the maximum value of the histogram corresponds to some fixed number of bars (~20).

An histogram is a representation of the distribution of the data. For our purposes, the histogram can be represented as a vector of integers. Each element (called bin) in this vector will represent the number of elements in the data that lie in a certain range. For instance, if the data is in the range [0, 1] and we have 10 bins, each bin will represent the number of elements in the ranges [0, 0.1], [0.1, 0.2], ..., [0.9, 1].

Example

Source code

```
$ ./histogram
|||||||||||||
||||||||||
|||||||||||||
||||||||
|||||||||||||||||
|||||||||||||
||||||||||
|||||||||||
```

Milestone

Write a function that generates a vector of uniform random numbers between 0 and 1, then print them.

Milestone

Write a function that takes a vector of numbers and returns a histogram (as a vector of ints).

Hint

- Start with a vector full of zeros.
- Find the bin that corresponds to each random number, increment by one the corresponding bin in the histogram.
- Map the range [0, 1] to the range [0, N] where N is the number of bins.

Advanced milestone

Generalize the histogram function so that it will work regardless of the range of the data.

Hint

- Find the minimum and maximum values of the input data.
- The <algorithm> header might help you, check its documentation.

Milestone

Write a function that will print the histogram. With a line for each bin. Use the character '|' to create the bars.
Try to issue a single call to cout per line.

Hint

- Normalize the histogram so that the maximum value corresponds to a fixed number of bars.
- You will need to find the maximum value in the histogram.
- You can concatenate strings in C++ using the "+" operator.

Advanced milestone

There is a really clean way to create the string that represents each bin.

Hint

- Look in the documentation for the constructor of string. Perhaps one of them will help you.

Advanced milestone

Try a random distribution that is not uniform, like the normal distribution.

Milestone

Write a test for your histogram function.

- One sanity check is to sum all the elements in the histogram and check that it is equal to the number of elements in the input vector.
- You can write another test by passing a manually crafted vector to your histogram function, for which you know the values histogram, and checking against that.

Make your program output an error message and exit if one of the tests do not pass.

Advanced milestone

If you are in for a challenge, try to extend your program to work in two dimensions. So that the input data consists of pairs of numbers (points in 2D space) and the histogram has bins in two directions.

Hint

- Representation will be more challenging, since now you need to print a heatmap (like a grid) using only ASCII characters.
  I will give you a couple approaches:
  - Use a set of characters to represent the intensity of the color, depending on the value of the bin. Like ·:-=+*#@$. You can also use UNICODE characters, which offer a lot of possibilities.
  - Print a grid of size nbinsx nbinsy of Unicode square characters (U+25A0). Use ANSI escape codes to change the color of the characters, so that blue corresponds to low values and red to high values. This one can get a little too complicated, so I am going to give you a couple links.