C++ for C programmers

Oct, 2022

Instructors: N. Hamid - R. Mahjoubi

What is C++

- C++ is a general-purpose
- Object-oriented programming language.
- created by Bjarne Stroustrup at Bell Labs circa 1980.
- C++ is very similar to C (invented by Dennis Ritchie in the early 1970s).
- C++ is so compatible with C that it will probably compile over 99% of C programs without changing a line of source code. Though C++ is a lot of well-structured and safer language than C as it OOPs based.

Why C++ ?

Stop coding C!

- C++ is a more structured and safer variant of C:
  There are very few reasons not to switch to C++.

- C++ (almost) contains C as a subset.
  So you can use any old mechanism you know from C
  However: where new and better mechanisms exist, stop using the old style C-style idioms.

C++ Scopes / Namespaces

Scopes

A portion of the source code where a given name is valid (Variable, Function, Class, Namespace…)

```cpp
#include <iostream>

using namespace std;

int main()
{
    int x = 10;

    cout << x << "\n";
    {
        int x = 20;
        cout << x << "\n";
    }
    cout << x << "\n";
    return 0;
}
```

From the terminal :

```
❯ ls
scope.cpp
❯ g++ scope.cpp
❯ ./a.out
10
20
10
```

Scopes

Variables are (statically) allocated when defined and freed at the end of a scope

```cpp
#include <iostream>

using namespace std;

int main()
{
    int x = 10;

    {
        int b[4];
        b[0] = x;
        cout << b[0] << "\n";
    }
    b[1] = x * 2;
    return 0;
}
```

From the terminal :

```
❯ g++ scope2.cpp
scope2.cpp:14:5: error: use of undeclared identifier 'b'
    b[1] = x * 2;
    ^
1 error generated.
```

Namespaces

- A namespace is a declarative region that provides a scope to the identifiers (the names of types, functions, variables, etc) inside it.

- Namespaces are used to organize code into logical groups and to prevent name collisions that can occur especially when your code base includes multiple libraries.

Namespaces

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

namespace first_space
{
  void func()
  {
     cout << "Space 1" << endl;
  }
}

namespace second_space
{
  void func()
  {
     cout << "Space 2" << endl;
  }
}

int main ()
{
  first_space::func();
  second_space::func();
  return 0;
}
```

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

namespace first_space
{
  void func()
  {
     cout << "Space 1" << endl;
  }
  namespace second_space
  {
     void func()
     {
        cout << "Nested space" << endl;
     }
  }
}

int main ()
{
  first_space::second_space::func();
  return 0;
}
```

C++ Input & Output

Input and output (I/O) streams

1. Input and output functionality is not defined as part of the core C++ language, but rather is provided through the C++ standard library (and thus resides in the std namespace).
2. When you include the iostream header, you gain access to a whole hierarchy of classes responsible for providing I/O functionality.

Streams:

- **stream** is just a sequence of bytes that can be accessed sequentially
- Over time, a stream may produce or consume potentially unlimited amounts of data.
- Typically we deal with two different types of streams. **Input streams** and **output streams.**

Input and output (I/O) streams

Input streams:

- Are used to hold input from a data producer, such as a keyboard, a file, or a network
- For example, the user may press a key on the keyboard while the program is currently not expecting any input. Rather than ignore the users keypress, the data is put into an input stream, where it will wait until the program is ready for it.

Output streams:

- Are used to hold output for a particular data consumer, such as a monitor, a file, or a printer
- When writing data to an output device, the device may not be ready to accept that data yet -- for example, the printer may still be warming up when the program writes data to its output stream. The data will sit in the output stream until the printer begins consuming it.

Input and output (I/O) streams

Input Output in C++:

- The **istream** class is the primary class used when dealing with input streams. With input streams, the **extraction operator** (**>>**) is used to remove values from the stream. This makes sense: when the user presses a key on the keyboard, the key code is placed in an input stream. Your program then extracts the value from the stream so it can be used.

- The **ostream** class is the primary class used when dealing with output streams. With output streams, the **insertion operator** (**<<**) is used to put values in the stream. This also makes sense: you insert your values into the stream, and the data consumer (e.g. monitor) uses them

- The **iostream** class can handle both input and output, allowing bidirectional I/O.

Input and output (I/O) streams

Input Output in C++:

- Standard streams in C++

A standard stream is a pre-connected stream provided to a computer program by its environment. C++ comes with four predefined standard stream objects that have already been set up for your use.

- ❏ cin -- an istream class tied to the standard input (typically the keyboard)
- ❏ cout -- an ostream class tied to the standard output (typically the monitor)
- ❏ cerr -- an ostream class tied to the standard error (typically the monitor), providing unbuffered output

Input and output (I/O) streams

Std::cout example:

cout.cpp

```cpp
#include <iostream> // for std::cout

int main()
{
    int x{ 5 }; // define integer variable x, initialized with value 5
    std::cout << x; // print value of x (5) to console
    return 0;
}
```

Input and output (I/O) streams

Std::cin example:

cin.cpp

```cpp
#include <iostream>  // for std::cout and std::cin

int main()
{
    std::cout << "Enter a number: "; // ask user for a number

    int x{ }; // define variable x to hold user input (and zero-initialize it)
    std::cin >> x; // get number from keyboard and store it in variable x

    std::cout << "You entered " << x << '\n';
    return 0;
}
```

Strings

C++ strings

What is string in c++ ?

One of the most useful data types supplied in the C++ libraries is the string. A string is a variable that stores a sequence of letters or other characters.

Syntax:

```
1 #include <iostream>
2 using namespace std;
3 int main()
4 {
5   string Str;
6   Str = "This is a string.";
7   return (0);
8 }
```

C++ strings

Why using c++ strings ?

The main advantage of using strings in c++, is that strings use dynamic memory allocation. This means you do not have to create your variable with a fixed size during initialization that you will not be able to change throughout the whole program. However, instead, a string can easily be resized, concatenated, shrunk, and manipulated during runtime.

Operations on strings

Basic Operations :

- Counting the number of characters in a string. The **length** method returns the number of characters in a string, including spaces and punctuation.

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

int main() {
  string str;
  str = "Lorem ipsum dolor sit amet, consectetur adipiscing elit.";
  cout << "Length of str : " << str.length()  << endl;

  return 0;
}
```

Operations on strings

Basic Operations :

- Accessing individual characters. Using **square brackets,** you can access individual characters within a string as if it's a char array. Positions within a string str are numbered from **0** through **str.length() - 1**. You can read and write to characters within a string using **[ ]**.
- Comparing two strings. You can compare two strings for equality using the **==** and **!=** operators. You can use **<**, **<=**, **>**, and **>=** to compare strings as well.

Operations on strings

Basic Operations :

- Appending to a string: C++ strings are wondrous things. Suppose you have two strings, **s1** and **s2** and you want to create a new string of their concatenation. Conveniently, you can just write **s1** + **s2**, and you'll get the result you'd expect. Similarly, if you want to append to the end of string, you can use the **+=** operator. You can append either another string or a single character to the end of a string.

```
#include <string>
#include <iostream>

using namespace std;

int main()
{
  string s1 = "Hello";
  string s2 = "world";

  string full_str = s1 + " " + s2;
  full_str += "!";
  cout << full_str << endl;
  return (0);
}
```

Operations on strings

Basic Operations :

- Searching within a string. The string member function **find** is used to search within a string for a particular string or character.
- Extracting substrings. Sometimes you would like to create new strings by extracting portions of a larger one. The **substr** member function creates substrings from pieces of the receiver string.
- Modifying a string by inserting and replacing. Finally, let's cover two other useful member functions that modify the receiver string. The first, **str1.insert(start, str2)**, inserts **str2** at position **start** within **str1**, shifting the remaining characters of **str1** over. The second, **str1.replace(start, length, str2)**, removes from **str1** a total of length characters starting at the position **start**, replacing them with a copy of **str2**.

Arrays

C style arrays

Fixed size arrays:

1. An array is an aggregate data type that lets us access many variables of the same type through a single identifier.
2. A **fixed array** (also called a **fixed length** array or **fixed size array**) is an array where the length is known at compile time

Syntax:

```
1   #include <iostream>
1 int main()
2 {
3     int prime[5]{}; // hold the first 5 prime numbers
4     prime[0] = 2; // The first element has index 0
5     prime[1] = 3;
6     prime[2] = 5;
7     std::cout << "The lowest prime number is: " << prime[0] << '\n';
8     return 0;
9 }
```

C style arrays

Fixed size arrays:

When declaring a fixed array, the length of the array (between the square brackets) must be a compile-time constant. This is because the length of a fixed array must be known at compile time.
Consider the following examples:

```
5 // using a non-const variable
4 int daysPerWeek{};
3 std::cin >> daysPerWeek;
2 int numberOfLessonsPerDay[daysPerWeek]{}; // Not ok -- daysPerWeek is not a compile-time constant!
1
  // using a runtime const variable
1 int temp{ 5 };
2 const int daysPerWeek{ temp }; // the value of daysPerWeek isn't known until runtime
3 int numberOfLessonsPerDay[daysPerWeek]{}; // Not ok
```

C style arrays

Passing fixed arrays to functions:

When passing an array as an argument to a function, a fixed array decays into a pointer, and the pointer is passed to the function:

```
1  #include <iostream>
1
2  void printSize(int* array)
3  {
4      // array is treated as a pointer here
5      std::cout << sizeof(array) << '\n'; // prints the size of a pointer, not the size of the array!
6  }
7
8  int main()
9  {
10     int array[]{ 1, 1, 2, 3, 5, 8, 13, 21 };
11     std::cout << sizeof(array) << '\n'; // will print sizeof(int) * array length
12
13     printSize(array); // the array argument decays into a pointer here
14
15     return 0;
16 }
```

C style arrays

Passing fixed arrays to functions:

Finally, it is worth noting that arrays that are part of structs or classes do not decay when the whole struct or class is passed to a function. This yields a useful way to prevent decay if desired.

C style arrays

Fixed size arrays:

A note on fixed size arrays:
Because fixed arrays have memory allocated at compile time, that introduces two limitations:

- Fixed arrays cannot have a length based on either user input or some other value calculated at runtime.
- Fixed arrays have a fixed length that can not be changed

In many cases, these limitations are problematic. Fortunately, C++ supports a second kind of array known as a dynamic array. The length of a dynamic array can be set at runtime, and their length can be changed.

C style arrays

Dynamic arrays:

C++ supports three basic types of memory allocation:

- **Static memory allocation** happens for static and global variables. Memory for these types of variables is allocated once when your program is run and persists throughout the life of your program.
- **Automatic memory allocation** happens for function parameters and local variables. Memory for these types of variables is allocated when the relevant block is entered, and freed when the block is exited, as many times as necessary.
- **Dynamic memory allocation**, the program is responsible for managing the memory.

C style arrays

Dynamic arrays:

To allocate an array dynamically, we use the array form of new and delete (often called new[] and delete[]):

```
#include <iostream>

int main()
{
    std::cout << "Enter a positive integer: ";
    int length{};
    std::cin >> length;
    int* array{ new int[length]{} }; // use array new.  Note that length does not need to be constant!
    delete[] array; // use array delete to deallocate array
    return 0;
}
```

C style arrays

Initializing dynamically allocated arrays

Prior to C++11, there was no easy way to initialize a dynamic array to a non-zero value (initializer lists only worked for fixed arrays). This means you had to loop through the array and assign element values explicitly.
However, starting with C++11, it's now possible to initialize dynamic arrays using initializer lists!

```
1   int fixedArray[5] = { 9, 7, 5, 3, 1 }; // initialize a fixed array before C++11
  1 int* array{ new int[5]{ 9, 7, 5, 3, 1 } }; // initialize a dynamic array since C++11
  2 // To prevent writing the type twice, we can use auto. This is often done for types with long names.
  3 auto* array{ new int[5]{ 9, 7, 5, 3, 1 } };
```

C style arrays

Resizing arrays

Dynamically allocating an array allows you to set the array length at the time of allocation. However, C++ does not provide a built-in way to resize an array that has already been allocated. It is possible to work around this limitation by dynamically allocating a new array, copying the elements over, and deleting the old array. However, this is error prone, especially when the element type is a class (which have special rules governing how they are created).

Consequently, we recommend avoiding doing this yourself.

Fortunately, if you need this capability, C++ provides a resizable array as part of the standard library called std::vector. We'll introduce std::vector shortly.

Introduction to std::array

STD::ARRAY

Fixed size arrays:

1. **std::array** provides fixed array functionality that won't decay when passed into a function
2. **std::array** is defined in the <array> header, inside the std namespace.

Declaration and initialization:

```
1  #include <array>
2
3  std::array<int, 5> myArray = { 9, 7, 5, 3, 1 }; // initializer list
4  std::array<int, 5> myArray2 { 9, 7, 5, 3, 1 }; // list initialization
5
```

STD::ARRAY

Passing std::arrays to functions:

Because **std::array** doesn't decay to a pointer when passed to a function, the size() function will work even if you call it from within a function:

```
13 #include <array>
12 #include <iostream>
11
10 void printLength(const std::array<double, 5>& myArray)
 9 {
 8     std::cout << "length: " << myArray.size() << '\n';
 7 }
 6
 5 int main()
 4 {
 3     std::array myArray { 9.0, 7.2, 5.4, 3.6, 1.8 };
 2
 1     printLength(myArray);
14
 1     return 0;
 2 }
```

STD::ARRAY

Range-based for-loops:

Because the length is always known, range-based for-loops work with std::array:

```
5 #include <array>
4 #include <iostream>
3
2 int main()
1 {
      std::array myArray{ 9, 7, 5, 3, 1 };
1
2     for (int element : myArray)
3         std::cout << element << ' ';
4 }
```

C++ vectors

C++ Exceptions

C++ Templates

C++ Iterators

C++ object oriented programming