Data Structures

Michele Riccardo Esposito

December 24, 2012

Contents

Chapter 1

Introduction

1.1 Let's get started

1.1.1 Purpose of the class

Welcome to the data structure class. During this class, you will learn the most used structures is computer science, as well as C++. This class will prove to be very useful to you for the following reasons:

1. Data structures are a must-know for every computer scientist and programmer.
2. 90% of interview questions for software engineering position are about data structures.
3. Will significantly improve your programming skill, and take your employability to the next level.
4. Give you insight on how things happen when you execute a program.

By the end of this class, you will have a thorough understanding of the most useful data structures. For every data structure that I will cover in this class, you will learn how to use them, how to implement them, and when to use them.

1.1.2 Structure

For most of the classes, you will be given a programming assignment, which I will refer to as **Machine Problem(MP)**. MPs are a wonderful way for you to prove your understanding of data structures. You will be asked to implement the structures, and use them in some significant way. I expect you to have some programming experience prior to this class. We will be moving fast, so it is important that you have some prior experience.

The MPs will be in C++ for the following reasons:

- One of the most influential programming language of all time. A must know for every computer scientist.
- Allows memory management. Very important when dealing with data structures.
- Lower level than dynamic languages. You will be able to understand many more things about how a computer program actually works.

You are not required to know C++ before this class, although it will help. During the first two lectures, I will be covering some of the most important aspects of C++. So, lets dive straight into C++, and lets take a look at some code.

1.1.3 Intro to C++

C++ was developed on top of C, in order to add Object Oriented features to C. C++ also comes with a useful collections of data structures and functions, which is called **Standard Template Library (STD)**. Lets dive our first C++ program:

```
1  // this is a comment, this line will be skipped
2  /*
3   * this is also a commend, but uses more lines
4   */
5  #include <iostream>  // import print functions.
6  using namespace std; // we will be using std functions
7
8  // main will be always executed first.
9  int main ( int argc, const char * argv[] )
10 { //  This is the entry point of our program.
11   cout << "Hello World" << endl; // print to console
12   return 0;
13 }
```

In order to compile this code, type in your terminal:

```
g++ -o hello helloWorld.cpp
```

and then run the executable **hello** by

```
./hello
```

Congratulations! You made your first C++ program. Lets now move to a more interesting example. Here is an implementation of a Fibonacci function:

```
1  int fibonacci(int n) {
2    if ( n == 0 || n == 1 )
3        return n;
4
5    int fib1 = 0;
6    int fib2 = 1;
7    int fib;
8
9    for ( int i = 2; i < n; i++ )
10   {
11       fib = fib1 + fib2;
12       fib1 = fib2;
13       fib2 = fib;
14   }
15   return fib;
16 }
```

Now, lets analyze this code:

1. Every **expression** in C++ needs to end with a semicolon.
2. Blocks of code need to be wrapped with curly braces.
3. On line *2*, the || operator means **logic or**.
4. C++ is a static typed language. That is, you have to declare a variable type before you use it. Notice, that the **fibonacci** function returns a value of type int. Likewise, all the variables defined have a associated type. C++ has the following primitive types:

Type name	**Translation**	**Values**	**Dimension in bit(s)**
bool	Boolean	true, false	1
char	Character	$[-2^4 \dots 2^4]$	8
short	Integer	$[-2^8 \dots 2^8]$	16
int	Integer	$[-2^{16} \dots 2^{16}]$	32
float	Floating Point		16
double	Floating Point		32
void	Null		

   Converting from one type to another is called **casting**.

5. The loop on line *9* can be decomposed as:

```
1      int i = 2;
2      while ( i < n )
3      {
4        /* do stuff */
```

```
5         i++;
6       }
```

where $i++$ is equivalent to $i = i + 1$ or $i+ = 1$.

You can also have multiple variable loops, such as

```
1 for ( int i =0, int j=3; i < n, j++, i ++ )
```

For every iteration, both j and i will increase. Here is a table of operators for C++:

Operator	**Meaning**	**Example**
++	Increase by one	$i++;$
−−	Decrease by one	$i--;$
&&	Logic and	i && $false$ will always return $false$
||	Logic or	i || $true$ will always return $true$
!	Logic negation	!0 == 1 will always return $true$

Also, we have the basic math operators, such as $+, -, *, /$.
In C++ there are two different types of arrays. The first type, are arrays that cannot change their length, and are called static arrays. The second type can change their length, and are called dynamic.
Here is an example on how to use a static array:

```
1  int array [5]; // creates array of 5. not initialized
2  int five  [5] = { 0, 1, 2, 3, 4 }; // you can initialize arrays this way
3  //int five [] = { 0, 1, 2, 3, 4 }; this is also valid
4
5  for( int i=0; i < 5; i++ )
6  {
7    cout << five[i] << "  "; // will print 0 1 2 3 4
8    array[i] = i; // initialize array
9  }
10 return 0;
```

1.1.4 First C++ assignment

Lets get some practice. Open with your favorite editor the file 'snippets.cpp'. You will find several functions that need to be completed. You may notice that I have already created test cases, so that you can verify correctness of your code. Use the command:

```
g++ -o snippets snippets.cpp
```

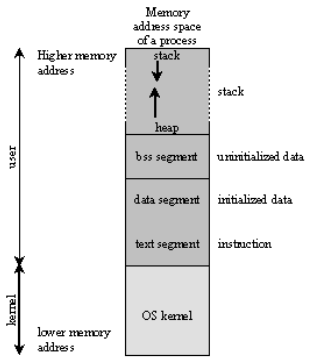

Figure 1.1: Process memory

and then run the executable **snippets** by

```
./snippets
```

Good luck!

1.2 C++ and memory management

In this lesson, you will learn how C++ manages memory, and you will gain understanding of how memory is managed inside your computer. Understanding these concepts is very important for programmers for several reasons:

- Allows you to optimize your program, avoiding to allocate useless memory.
- Understand what goes under the hood in dynamic programming languages.
- Can avoid security issues.
- Learn our first data structure!

1.2.1 Process memory

Every process inside a computer has a certain amount of memory, which is granted to the process by the operative system. Some of that memory will be reserved to keep some information about the process. That memory cannot be accessed, and is reserved to the operative system. Then, we have that is called the **stack**. Look at Figure **??** to get an idea of the information:

Stack

The stack is used in order to keep track of the location where we are inside our process. Imagine that you have a certain number of tasks that you need to do. For every task, we receive a post-it, with some information about the task. A task may require you to have to complete a different task before you may be able to solve the previous one, and for every task you need to store some information, relative to the task. An excellent way to manage this information is to use a pile. For every task that you receive we will push it on top our pile, along with the informations that we need in order to complete the task. If the task will require some other task to be completed, then we will begin working on that task, and push it on top of our pipe. Once a task is completed, then we remove it from the top of our pile. When, the pile will be empty again, then our program will be over!
In computers programs, things work in the same way. Instead of tasks we have functions, and our pipe is what we refer to as a stack, an abstract data structure. The information that we need in our process are variables, that we need to keep track of in order to complete our tasks. Lets take a look at an example:

```
1  int task ( int sum )
2  {
3    int total = 0;
4    for ( int i =0; i < sum; i++ )
5      total +=i;
6    return total;
7  }
8
9  int main ( int argc, const char * argv[] )
10 { //  This is the entry point of our program.
11   int sum = 5;
12   int total = task( sum );
13   total    += task( sum ); // += will add to the existing value
14   return 0;
15 }
```

Now, lets take a look at what happens inside our stack when we execute the program. We start with main inside the stack. Inside the space for main, there will be enough space for the variables *sum*, and *total*. On line *13*, the variable sum will take the value 5. Then, the function *task* will be called.
New space will be asked for on the stack, so that we can fit the variables that we have inside *task*. The value of *sum* will be copied over, and the function *task* will have its own copy. Likewise, *task* will have its own version of *total*. These are **local variables**, so they only exist in the scope of the function. You cannot access the local variables of *main* from the function *task*. Once the first call to *task* will be completed, we will copy the value of *total* to the main function, and the space on the stack that was reserved to *task* will be erased. Just like in our analogy we used to throw away tasks once they were completed. This is how the memory stack works!

Heap

There is one more complication to how memory management works. Imagine, we are in the middle of on of our functions/tasks. All of a student, we discover that we need more memory than we thought before. Where do we get the memory from? We cannot change the size of memory that we have on the stack, yet we need more memory. This is where dynamic memory comes in. If we do not know the amount of memory that we are going to use for a certain task, then we are going to use something called the **heap**. The heap is another memory segment in the process memory. It is used to allocate dynamic memory. You can allocate and free memory on the heap as you wish, without many limitations. In order to create memory on the heap in C++, you have to use the keyword new. Before we can look at an example, we have to look at something called pointers.

1.2.2 Pointers and reference

In C++, every variable has two attributes:

1. Type: E.x. int, bool.
2. The location of the variable in memory, which is a hexadecimal address. Eg. $0x301ca39d$.

When we use a variable, C++ goes into memory and looks at the value of that variable. Whenever we refer to a variable, C++ automatically lookup the value that it holds in memory. However, you can also lookup the address by using the **address** operator, **&**. Lets take a look at an example:

```
1 int look  = 5;
2 cout <<  look << endl; // will print 5
3 cout << &look << endl; // will print some address location.
```

Now, if you have an address, you can either store it, using another variable, or look up the memory inside the value. A variable that stores an address in memory is called a **pointer**. Lets continue looking at the previous example:

```
1 int * pointerToLook = &look;
2 cout << * ( & look )    << endl; // will print 5
3 cout << * pointerToLook << endl; // will print 5
4 cout <<   pointerToLook << endl; // will print some address. Equivalent to & look.
```

So, you can create a pointer variable using the star operator before the name of the variable, and then you can lookup the value inside memory using the star operator.

1.2.3 Allocating memory in C++