Data Structures

Michele Riccardo Esposito

December 26, 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
```

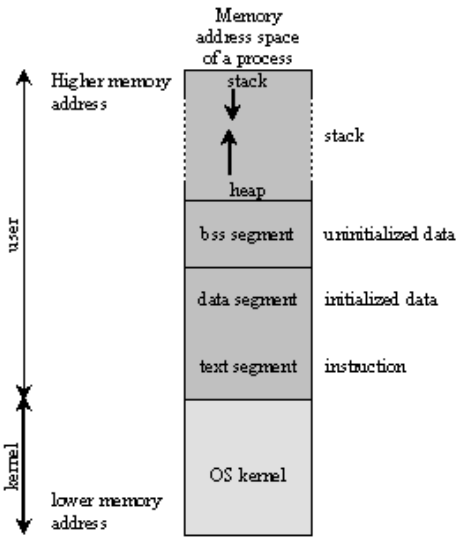

Figure 1.1: Process memory

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 1.1 to get an idea of how process memory looks like:

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 make 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. How are we going to keep track of all the post-its that we will write?
An excellent way to manage this information is to use a pile. For every post-it that we make, we will push it on top our pile. If the task will require some other task to be completed, then we will begin working on that task, and push the relative post-it on top of our pile. Once a task is completed, then we remove the post-it from the top of our pile, and resume working on the task with the previous post-it. 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 pile 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. The post-its are the chunks of memory that we use to store that information inside our process. 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 post-its 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 one 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**. You can lookup values inside pointers by using the **star operator**. Using the star operator to lookup the value of a memory address is called **dereferencing** a pointer. Lets continue looking at the previous example:

```
int * pointerToLook = &look;
cout << * pointerToLook << endl; // will print 5
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. Note that the two operatons, are reverse to each other. In fact, if you take the address and then dereference, it will be like doing nothing.

```
cout << *(&look ) == look << endl; // will print true
```

You have to be careful whenever you use the star operator. If you follow an address that doesn't exist in memory, then you will get a segmentation fault. That is, the computer will complain that the memory address that you have asked does not exist, therefore it will terminate the program. You have to be careful not to use uninitialized pointers, or pointer that point to NULL. Here is an example of segfaults:

```
int * a;
int * b = NULL;
* a = 3; // segfault
int c = * b; // seg fault
```

You can use pointers in two different ways. You can either use them to point at existing memory, or make them point at new memory you create for them. Both these cases can be very useful. Lets look at some examples:

```
int a = 5;
int * aReference = & a;
cout << a << endl; // prints 5
cout << * aReference << endl; // prints 5
* aReference = 8;
cout << a << endl; // prints 8
cout << * aReference << endl; // prints 8
a = 0;
cout << a << endl; // prints 0
cout << * aReference << endl; // prints 0
int c = 39;
aReference = & c;
cout << a << endl; // prints 0
cout << c << endl; // prints 39
cout << * aReference << endl; // prints 39
```

Using pointers this way is very useful in case you want to keep allocating new memory for values. Say for example that you have a very large object, and you want to pass it inside a function. Copying the object would be very expensive. With pointers instead, you can simply pass in the address, and then you are done. There are three ways to pass in an object. By value, by reference and by pointer.
By value, it means that you copy over the value of the object you are passing in. This works very well with small object, such as primitive types. The object that is passed in will not change if you change it inside the function, because it is a distinct copy of the value that you have passed in.
Passing in values by reference and by pointers instead will change the value of the object that you pass inside. Both these two methods do not copy the object, but simply pass inside the function the address of the memory object you are passing inside the function. The two methods are fundamentally the same, they are only different in their syntax. Sometimes is more convenient to use reference, other pointer.

```
1  int byValue ( int temp )
2  {
3    temp += 5;
4    return temp - 8;
5  }
6
7  int byReference ( int& temp )
8  {
9    temp += 5;
10   return temp - 8;
11 }
12
13 int byPointer ( int * temp )
14 {
15   * temp += 5;
16   return * temp -8;
17 }
18
19 int main ( int argc, const char * argv[] )
20 {
21   int total = 5;
22   cout << byValue     ( total )   << endl; // print 2
23   cout << total                   << endl; // total not changed
24   cout << byReference ( total )   << endl; // print 2
25   cout << total                   << endl; // total changes to 10
26   cout << byPointer   ( & total ) << endl; // print 7
27   cout << total                   << endl; // total changes to 15
28 }
```

Now that we have a good understanding of pointers, lets see how to allocate new memory on a newly declared pointer.

1.2.3 Allocating memory in C++

1.3 Mathematics review

In this lesson, we will go over some of the basic mathematics concepts that we need in order to tackle data structures. The first in Big-O notation, which is very important in computer science, as well as in programming. Then, I will be talking a bit about induction and recursion. I want to get the following things out of this lesson:

- Have a great understanding of what Big-O is.
- Be able to do Big-O analysis quickly.
- Understand the concept of mathematical induction.
- Have a better understanding of recursion.

1.3.1 Big-O notation

Big-O notation is fundamental in computer science. It is used in order to evaluate the runtime or space complexity of an algorithm. Significant parts of computer science theory were developed before computers even existed. However, computer scientist needed a way to be able to tell how fast algorithms would be running. So they invented Big-O notation. Here is the mathematical definition of Big-O:

> $f(x)$ is $O(g(x))$ if and only if there are positive real numbers c, k such that $0 \leq f(x) \leq cg(x)$, for every $x \geq k$.

What this means is that $f(x)$ will be $O(g(x))$ only if they grow to infinity with the same speed. For example, if $f(x) = 4x$, then $f(x)$ is $O(n)$, because if we let $c >= 4$, then $0 \leq f(x) \leq cg(x)$. All Big-O is saying is: ignore constants, and look at the higher term of a function. For example, the function $h(x) = n^3 + 40942n^2 + \sqrt(n)$ is order $O(n^3)$. Now, when we look at the speed of functions in real life, constants do matter. However, in computer theory we are not so concerned about them, as the order of growth of a function is much more important. Doing Big-O analysis is not hard, it just requires some practice. Lets take a look at some examples:

```
1 int arraySum ( int * array, int size )
2 {
3   int total = 0;
```

```
4   for( int i = 0; i < size; i ++ )
5     total += array[i];
6   return total;
7 }
8
9 int halfArraySum ( int * array, int size )
10 {
11   int total = 0;
12   for( int i = 0; i < size/2; i ++ )
13     total += array[i];
14   return total;
15 }
```

When you want do Big-O analysis, the first thing you want to do, is to figure out what is the variable in the function. In the two functions above, the variable is *size*. In order to sum an array, we need to iterate through every element of the array, so that we can add it to total. So, $arraySum$ requires iterations $i(n) = n$, thus has order $O(n)$. However, also $halfArraySum$ has order $O(n)$, as it requires iterations $i(n) = \frac{n}{2}$. Remember? Constants do not matter. Imagine that we give an infinitely long array to our functions. Then, the two functions will take infinitely the same time. Lets look at some other examples:

```
1 /* ADD MORE EXAMPLES */
```

Chapter 2

Linear Data structures

2.1 Abstract Data Type

An Abstract Data Type (ADT) is a model for data structures. What it allows us to do, is to focus on the structure itself instead of the implementation. This is useful, because like this we can also look at different implementations of the same structures. Almost every ADT that we will be looking at, will be an implementation of a **dictionary**. That is, we will have a pair *key*, *value* insert inside the data structure. Then, we will be retrieving the values by looking up the associated *key*. A dictionary is very useful for several reasons.

1. Allows you abstract between key and value.
2. Allows many operations on data, such as sorting keys of finding min/max keys.
3. Allows faster insert/retrieval.

2.2 Linked Lists

The first ADT that we will be looking at are Linked Lists.