Data Structures and Object Oriented Programming

Waqar S. Qureshi

NUST College of Electrical & Mechanical Engineering.
waqar.shahid@alumni.ait.asia

October 4, 2017

Table of contents

File Stream

C++ Files and Streams

* iostream standard library provides cin and cout methods to read input and write to output respectively
* Now, we will learn how to write and read from a file using fstream library
* In the examples, we will also use sstream, iostream, and learn how to generate error message when reading files

Header files to be included for reading and writing from files are:

```
#include <fstream>
#include <iostream>
#include <sstream>
#include <string>
```

Streams

Streams are the C++ ways of interacting with files, keyboard, screen, and also strings

Stream	**target**	**header file**
`cin`	keyboard	`iostream`
`cout,cerr`	screen	`iostream`
`ifstream`	File to read	`ifstream`
`ofstream`	File to write	`ofstream`
`stringstream`	String	`sstream`

Table : Different header files required for `stream`

Steams: Important points

A few things that are worth remembering about reading from an input stream:

1. When reading from a stream (say, cin, or a ifstream myinput), you can use `cin.fail()` (or `myinput.fail()`) to check whether the read succeeded. When one read fails (and the `fail` flag is set), it will remain set until you explicitly reset it with `cin.clear()` or `myinput.clear()`. Until then, all further reads will fail, since C++ assumes that until youve fixed the source of the problem, all incoming data will be unreliable now.

2. Remember that >> reads until the next white space, which includes spaces, tabs, and newlines.

1. If you want to also read spaces, then instead of using >>, you should use getline, such as `cin.getline()` or `myinput.getline()`. It lets you specify a string to read into, a maximum number of characters to read, and character to stop at (default is newline), and will then read into the string. That string can later be further processed.
2. If you want to "clear" off the remaining characters in a stream so they dont cause issues later, you can use the `ignore()` function, which will ignore the next n characters

Example: String Stream

In the example note the use of `string` and `char`

Example: sstream

```
void StringStreamExample(){
   int first ;
   char second ;
   string third ;
   stringstream ss ;

   ss<< "1: Minecraft";
   ss>>first;
   ss>>second;
   ss>>third;
   cout<<first<<endl<<second<<endl<<third<<endl;
}
```

Example: File Stream

In the example note the use of `cerr` and `good()` function

Example: fstream

```
void FileStreamExample () {
   ofstream myFile1 ;
   myFile1.open("game.txt") ;
   myFile1<<"1: Minecraft"<<endl ;
   myFile1.close() ;
   ifstream myFile2("game.txt");
   string line ;
   if(!(myFile2.good())){
      cerr << "Error Reading File : "<< endl;
   }
   else{
getline(myFile2,line);
        cout<<line<<endl;
   }
   myFile2.close() ;
}
```

Example: String Stream

In the example note the use of `cerr` and `good()` function

Example: fstream

```
void FileStreamExample () {
   ofstream myFile1 ;
   myFile1.open("game.txt") ;
   myFile1<<"1: Minecraft"<<endl ;
   myFile1.close() ;
   ifstream myFile2("game.txt");
   string line ;
   if(!(myFile2.good())){
      cerr << "Error Reading File : "<< endl;
   }
   else{
getline(myFile2,line);
        cout<<line<<endl;
   }
   myFile2.close() ;
}
```

Example: Reading and Writing from File and Standard input

```
void ExampleFileStream () {
   char data[100];
   int age;
   char InputFile[100];
   // Get the name of the file
   cout << "Enter The file Name: ";
   cin.getline(InputFile, 100);

   // open a file in write mode.
   ofstream outfile;
   outfile.open(InputFile);

   cout << "Writing to the file" << endl;
   cout << "Enter your name: ";
   cin.getline(data, 100);

   // write inputted data into the file.
   outfile << data << endl;
   cout << "Enter your age: ";
   cin >> age;
   cin.ignore();

   // again write inputted data into the file.
   outfile << age << endl;
   // close the opened file.
   outfile.close();
```

```
   // open a file in read mode.
   ifstream infile;
   cout << "Enter the name of
   the File to read" << endl;
   cout << "Filename: ";
   cin.getline(InputFile, 100);
   infile.open(InputFile);
   if(!(infile.good())){
cerr<<"Error: File does not exist!"<<endl;
   }
   // write the data at the screen.
   else{
      cout << "Reading from the
      File: " <<InputFile<< endl;
      infile >> data;
  cout << data;
  infile >> data;
  cout << " "<<data;
  infile >> data;
  cout << " "<<data;
  infile >> age;
  cout << " :"<< age << " years"<<endl;
   }
   // close the opened file.
   infile.close();
   return 0;
}
```

Dynamic Memory Allocation

Dynamic Memory Allocation

Fixed arrays are statically allocated in stack memory e.g. `int a[100]` or `double b`

We may need an array or memory chunk for which we do not know the size during compilation

Such memory allocation is done in `C++` or `C` using dynamic memory allocation.

The memory is allocated on the heap space instead of the stack space as the size is unknown before the execution of the program.

Dynamic memory allocation

Static allocation	**Dynamic allocation**
Size must be known at compile time	Size may be unknown at compile time
Performed at compile time	Performed at run time
Assigned to the stack	Assigned to the heap
First in last out	No particular order of assignment

Table : Differences between statically and dynamically allocated memory

Dynamic memory allocation

In order to dynamically allocate and deallocate memory, there are two pairs of functions, one in C-style and one in C++ style

In C, the function for allocating memory is `malloc`, and for deallocation `free`

In C++, the functions are `new` and `delete`

C style, is little closer to the actual low-level implementation

Let us see few examples in the next slides

C-style dynamic memory allocation

The function `void* malloc (unsigned int size)` requests `size` bytes of memory from the operating system and returns the pointer to that location as a result

If for some reason, the OS failed to allocate the memory (e.g., there was not enough memory available), `NULL` is returned instead

The function `void free (void* pointer)` releases the memory located at pointer for reusing

Example: dynamic size of array

```
int n;
int* b;
cin >> n;
b = (int*) malloc(n*sizeof(int));
for (int i=0; i<n; i++)
cin >> b[i];
```

C-style dynamic memory allocation

Using `sizeof(int)` is much better than hard-coding the constant 4, which may not be right on some hardware now or in the future

Because `malloc` returns a `void*`, and we want to use it as an array of integers, we need to cast it to an `int*`

Another thing to observe here is that we can reference b just like an array, and we write b[i]

If we wanted to write b[i] in a complicated way by doing all the pointer arithmetic by hand, we could write instead
`*((int*) ((void*) b + i*sizeof(int)))` [1]

To return the memory to the *OS* after using memory, we use the function `free`, as follows:

```
free(b);
b = NULL;
```

[1] we do not type, this is just to make sure you understand!

C++ style dynamic memory allocation

C++ provides the `new()` and `delete()` functions that provide some syntactic sugar to C-style `malloc()` and `free()`

Basically, they relieve you from the calculations of the number of bytes needed,and provide a more "array-like" syntax

```
int n;
int *b;
cin >>n;
b = new int[n];
*p = new int;
```

`new` figures out by itself how much memory is needed, and returns the correct type of pointer

To release memory, the equivalent of `free` is the `delete` operator, used as follows:

```
delete [] b;
delete p;
```

Memory leaks and Garbage collection

Following is an example of what can go wrong while allocating dynamic memory

```
double *x;
...
x = (double*) malloc(100*sizeof(double));
...
x = (double*) malloc(200*sizeof(double)); // We need a bigger array now!
...
free(x);
```

The above code will create a memory leak. A better version of the code above would be as follow:

```
double *x;
...
x = (double*) malloc(100*sizeof(double));
...
free(x);
x = NULL;
x = (double*) malloc(200*sizeof(double));
...
free(x);
```

Recursion

Recursion

What is recursion?

The adjective recursive means "defined in terms of itself".

There are two types of recursions: "direct" and "indirect"

Four question for constructing recursive solutions:

1. How can you define the problem in terms of a smaller problem
2. How does each recursive call diminish the size of the problem
3. What instance of the problem can serve as the base case
4. As the problem size diminishes, will you reach this base case

Recursion toy example

Which of the following are correct?

```
int iterativeFactorial(int n) {
   int p=1;
   for (int i=1;i<=n;i++)
      p*=i;
   return p;
}

int recursiveFactorial (int n) {
   if(n==1) return 1;else return (n*recursiveFactorial(n-1));
}

int EMEfactorial (int n) {
   if(n==1) return 1;else return EMEfactorial(n);
}

int NustFactorial (int n) {
   return n*NustFactorial(n-1);
}
```

Linked List

How much space should we reserve/allocate

Dynamically sized arrays give us a partial solution for allocating memory size for an array of data

How large should be the array at run time when we declare it?

How large do you think that Facebook should have made its user array when it started

If you have more and more customers arriving over time, it will be very hard to guess the right size

Introduction to Linked list

Many data structures dont need to know the required number of elements beforehand rather, their size can dynamically change over time

The easiest such data structure is called the linked list

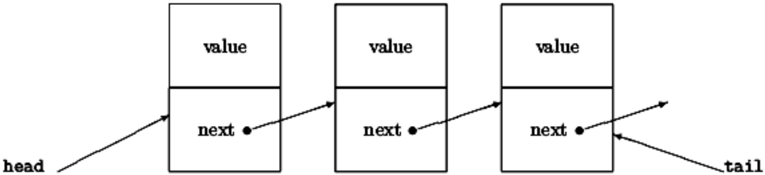

Figure : Basic illustration of a linked list

A series of nodes where each one points to the next one in memory, and each node contains a piece of data

Linked lists can be made as long as we want without declaring an initial size first

Linked List

Each node in the linked lists contains data (such as an int, string, etc.) as well as a redpointer to the next nodeof the same type

We will build a linked list of integers.

In order to keep track of these two elements, we create a `struct` which we call `Item`

Example:Every `item` has an `int` value and a pointer to next element

```
struct Item{
    int value;
    Item *next;
    Item (int val, Item *n){
        value = val; next = n;
    }
}
```

Linked List: Example

The function `Item` we declare inside the `struct` is used to make initialization easy

```
Item* p = new Item (5, NULL);
```

instead of

```
Item *p = new Item;
p->value= 5;
p->next = NULL;
```

To access the first element of the list, we need a head pointer to the first Item

If we lose track of this, the rest of the list can no longer be accessed

Linked list operations

Unlike arrays link list do not provide functionality to directly access its element

The important operations that we like the link list to support are:

1. to be able to add elements to our list
2. to be able to remove elements from our list
3. to be able to traverse the entire list

Abstract Data Types

References

References

C++ Overview (2017)
Website
https://www.tutorialspoint.com

The End