Computer Science Courses Notes

Ahmed Gamal Eltokhy

Sat, 13 Feb. 2021

Contents

I Fundamentals of Computing II **1**

1 Before Midterm 1 **2**

1.1 Review . . . 2

1.1.1 char, string . . . 2

1.1.2 enum . . . 2

1.2 Let's improve our Si by creating a neat code . . . 2

1.3 Static Shit . . . 4

1.4 Classes Notes . . . 6

1.5 Operators Overloading . . . 6

1.6 'this' operator . . . 9

1.7 Copy Constructor . . . 10

1.8 Templates . . . 10

1.9 Inheritance . . . 11

1.9.1 Public Inheritance . . . 11

Part I
Fundamentals of Computing II

1 Before Midterm 1

1.1 Review

1.1.1 char, string

cctype.h Includes tolower(char x), toupper(char x), isupper(char x), is alpha(char x), etc.

string
str.length() returns length, str.at(i) return the i^{th} character of string,
str.erase(i, n) erases n characters after the index i,
str.substr(i, n) returns substring of n characters after the i^{th} character,
str.replace(i, n, newstr) replaces n characters i^{th} with the string newstr,
str.empty() check if a string is empty,
str.swap(str2) swap str with str2,
str.find(char/ str) return char's index, or substring's first character index. If not found return length,
str.assign(str2, i, n) assign n characters starting from the i^{th} in str2 to str,
str.insert(i, str2) add str2 to str after the location i in str,
str.append(str2) append str2 to str, str.c_str() returns a C_style char*.
atoi(str.c_str()) returns string numeric (integer), atol does long, atof float

1.1.2 enum

See this example

```
enum day {Sun, Mon, Tue, Wed, Thu, Fri, Sat};
day x;
x = Fri;
cout << x;    //5
if (x == Sunday) cout << "Sunday";

enum nums{one = 1, two, three, four};
nums x = one; // x = 1
nums y = two; // y = 2
```

You can now guess that bool is an enum of true and false.

1.2 Let's improve our Si by creating a neat code

function.h

```
using namespace std;

class Student {
private:
   char *name;
```

```
   double **data;

public:
   double computeAnyThing();
};

//Hi you're using vim and no one fix your shit, #include "function.cpp"
//You can also \$ g++ -c *.cpp\ or make a fucking MakeFile
```

function.cpp

```
#include "function.h"
double Student::computeAnyThing() {
   return 9.0;
}
```

main.cpp

```
#include <iostream> //Any Library You Will use
#include "function.h"

int main {
   Student S_1;
   S_1.computeAnyThing();
   return 0;
}
```

1.3 Static Shit

Static elements are allocated storage only once in a program lifetime in static storage area. And they have a scope till the program lifetime.

```
void demo() {
    static int count = 0;
    cout << count << " ";
    count++;
}

int main() {
    for (int i=0; i<5; i++)
        demo();
    return 0;
}
// Output: 0 1 2 3 4
```

```
class Abc{
    int i;
    public:
   Abc(){
        i=0;
        cout << "constructor";
    }
   ~Abc(){
        cout << "destructor";
    }
};

void f(){
    static Abc obj;
}

int main()
{
    int x=0;
    if(x==0)   {
        f();
    }
    cout << "END";
}
//Output will be Constructor END Destructor
//This happens because lifetime of an object is same of the program
```

Static data members are shared between the objects and must be initialized explicitly outside the class not to get linker error. Also, they can not redefined by user.

Static functions can be called without any object.

```
class Test {
    static int x;
public:
    Test() { x++; }
    static int getX() {return x;}
};

int Test::x = 0;

int main() {
    cout << Test::getX() << " "; // 0
    Test t[5]; // iterates on the constructor of test 5 times
    cout << Test::getX(); // 5
}
```

A static function is a special type of function which is used to access only static data, any other normal data cannot be accessed through static function.

1.4 Classes Notes

If no explicit destructor or in a class, compiler create a empty destructor. Constructors and destructors are called implicitly by compiler

Order of calls

1. When execution enters scope of object constructor is called
2. When execution exits scope of object destructor is called or exit is called
3. Destructor is not called if program ends with abort

A const object is not modifiable, can only call const member functions can not modify an object. Const objects can not have setters or non const getters as well.

Constant functions do not modify objects, and they are declared as:
int getNum() const {return num;}
Note that non-const cosntructor can initialize a const object.
In short, const member functions can be used for all class objects, but must be used for const class objects.
Const data members must be initialized using initializer list.

```
class Increment{
   int cnt;
   const int val;
   public:
      Increment(int c=0; int i=1):
      cnt(c), // Initializer list can be done for normal integers
      val(i){ // But must be done for const integers
         //empty body
      }
}
```

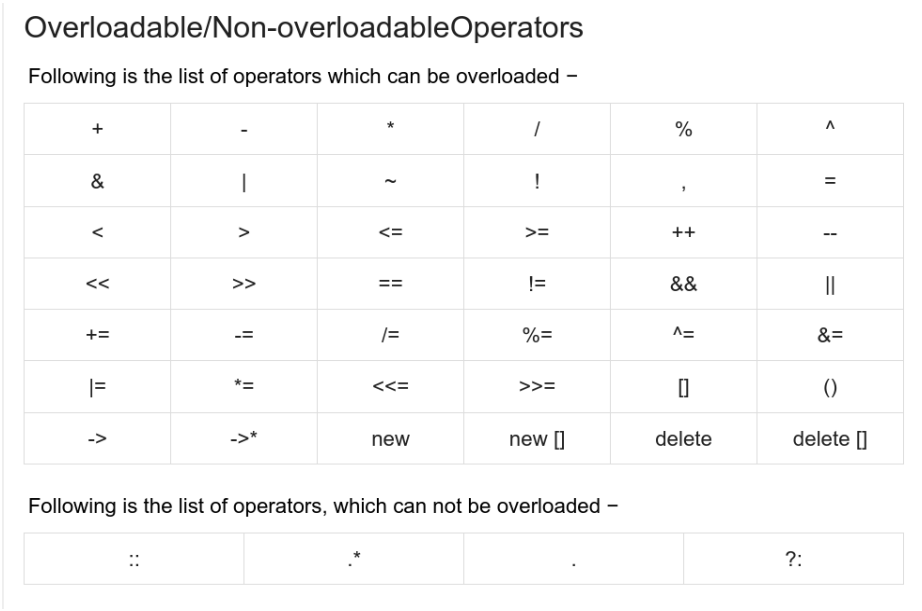
1.5 Operators Overloading

You can redefine or overload most of the built-in operators available in C++. Thus, a programmer can use operators with user-defined types as well.

```
class Box {
   public:
      double getVolume(void) {
         return length * breadth * height;
      }
      void setLength( double len ) {
         length = len;
      }
      void setBreadth( double bre ) {
         breadth = bre;
```

```
      }
      void setHeight( double hei ) {
         height = hei;
      }

      // Overload + operator to add two Box objects.
      Box operator+(const Box& b) {
         Box box;
         box.length = this->length + b.length;
         box.breadth = this->breadth + b.breadth;
         box.height = this->height + b.height;
         return box;
      }

   private:
      double length;    // Length of a box
      double breadth;   // Breadth of a box
      double height;    // Height of a box
};

// Main function for the program
int main() {
   Box Box1;              // Declare Box1 of type Box
   Box Box2;              // Declare Box2 of type Box
   Box Box3;              // Declare Box3 of type Box
   double volume = 0.0;   // Store the volume of a box here

   // box 1 specification
   Box1.setLength(6.0);
   Box1.setBreadth(7.0);
   Box1.setHeight(5.0);

   // box 2 specification
   Box2.setLength(12.0);
   Box2.setBreadth(13.0);
   Box2.setHeight(10.0);

   // volume of box 1
   volume = Box1.getVolume();
   cout << "Volume of Box1 : " << volume <<endl;

   // volume of box 2
   volume = Box2.getVolume();
   cout << "Volume of Box2 : " << volume <<endl;

   // Add two object as follows:
   Box3 = Box1 + Box2;

   // volume of box 3
   volume = Box3.getVolume();
```

```
cout << "Volume of Box3 : " << volume <<endl;

return 0;
/* Output:
*  Volume of Box1 : 210
*  Volume of Box2 : 1560
*  Volume of Box3 : 5400
*/
```



Overloadable/Non-overloadableOperators

Following is the list of operators which can be overloaded −

+	-	*	/	%	^
&	|	~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	&=
|=	*=	<<=	>>=	[]	()
->	->*	new	new []	delete	delete []

Following is the list of operators, which can not be overloaded −

::	.*	.	?:

Figure 1: Operators that can and cannot be overloaded, by tutorialspoint.com

1.6 'this' operator

The compiler supplies an implicit pointer along with the names of the functions as 'this'. The 'this' pointer is passed as a hidden argument to all nonstatic member function calls and is available as a local variable within the body of all nonstatic functions. The 'this' operator can be used in some situations as:

Similarity in the names of local variable and class member

```
class Test {
private: int x;
public:
   void setX (int x) {
       // The 'this' pointer is used to retrieve the object's x
       // hidden by the local variable 'x'
       this->x = x;
   }
   void print() { cout << "x = " << x << endl; }
};
```

Returning a reference to the object called

```
/* Reference to the calling object can be returned */
Test& Test::func () {
   // Some processing
   return *this;
}
```

When a reference to a local object is returned, the returned reference can be used to chain function calls on a single object.

```
class Test {
private:  int x, y;
public:
  Test(int x = 0, int y = 0) { this->x = x; this->y = y; }
  Test &setX(int a) { x = a; return *this; }
  Test &setY(int b) { y = b; return *this; }
  void print() { cout << "x = " << x << " y = " << y << endl; }
};
int main() {
  Test obj1(5, 5);
  // Chained function calls. All calls modify the same object
  // as the same object is returned by reference
  obj1.setX(10).setY(20);
  obj1.print();
  return 0;
}
```

1.7 Copy Constructor

In case of copy constructor, you can copy without having a getter.

```
//Here is passed by reference to avoid syntax error
className::className(const className &s_1) {
   ID = s_1.ID //id here is private
   // Let's assume you allocated memory here.
   for(int i=0; i<2 i++) {
      for(int j=0; j<2) {
         *(*(some_2D_arr+i)+j) = *(*(s_1.some_2D_arr+i)+j);
      }
   }
}
// We don't 'className s_2(s_1)' cuz fuck shadow copying
// Shadow copying if you forget is when you directly copy directly
    'p_1=p_2' while deep copying is '*p_1=*p_2' that dereference first
// So, it will make confusion in pointers as it'll point at a wrong array
```

1.8 Templates

This concept is so fucking simple, it's basically to make types of functions agnostic. Also, the syntax is fairly easy

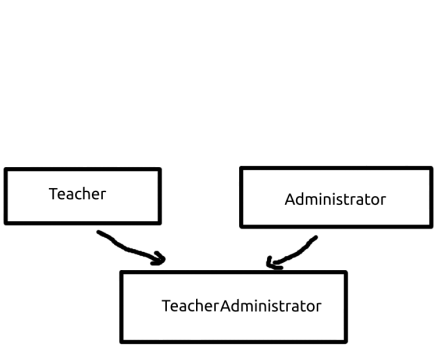
```
template <class X>
X absolute(X n) {
   n<0 ? return (-n) : return n;
}
// Multiple types in a template
template <class T_1, class T_2>
T_1 function(T_2 a, T_2 b) {
   return doAnyShit(a,b);
}
```

There is some situations as template specialization for some types (each type has its own functions in the class), and static members. For the second situation, each specialization has its own copy of static data members and member functions, and they are shared between all objects. For the first situation, I'll add an example here

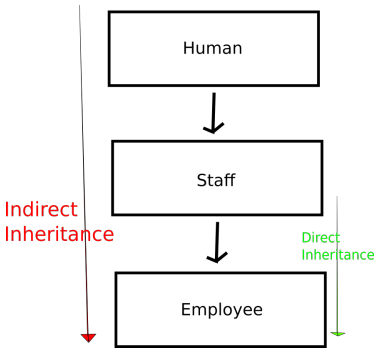
1.9 Inheritance

It's basically copying some data members or member functions from a **base class** to a new **derived cass**.

For instance, if we have a class names **administratorTeacher** that inherits from **Teacher, Administrator** classes, that is multiple inheritence. While if some class named **staff** that inherits one other class called **employee**, this is single inheritance. Also, if two classes inherit directly, this is called **Direct Inheritance**, but there are some inherited things from layers that were alreade inherited before, that is **Indirect Inheritance**.



(a) Multiple Inheritance



(b) Direct Inheritance

Base Class	Derived Class Access (Type of Inheritance)	Derived Class Member
private protected public	: private : private : private	inaccessible private private
private protected public	: public : public : public	inaccessible protected public
private protected public	: protected : protected : protected	inaccessible protected protected

Figure 3: Types of Inheritance

1.9.1 Public Inheritance

You can not change private data members, but you can use only the public setter functions.

However, you can use "protected" instead of private. It let the function be private except for the derived classes.

```
class vehicle { //base class
   protected:
   //If it was private, price wouldn't have been able to be played with.
      int price;
   public:
      vehicle(int a=0) {price = a;}
      // The constructor itself does not get derived, but the shit in it
          gets executed, as allocating memory or setting some variables
          for some reason.
};
class car:public/*inheritance type*/ base { //derived class
   public:
      int getEGPPrice(){return price*15;}
};
```

Note that You can call a function, for instance print() from the base class and add some parameters to it.

```
void derivedClass::print() {
   //You Can Print any extra code here
   baseClass::print();
}
```

List Initializers You can use initializer list in an inherited class for constructor initialization from the base class. You can see it in this example

```
class Base {
    int n;
};

class Base_2{
   int n2;
}

//EXAMPLE FOR MULTIPLE INHERITANCE AS WELL
class derived : public Base, Base_2
{
    unsigned char x;
    unsigned char y;
    std::string s;

    derived ( int x )
      : Base { 1 }, // initialize base classes
```

```
      Base_2{ 2 },
       x ( x ),     // x (member) is initialized with x (parameter)
       y { 0 },     // y initialized to 0
   {}               // empty compound statement

};
```

Note that you can give the address of a derived class to a pointer of the base class. However, you'll run by the issue that if you call a function, print() for example, which is overwritten in the derived class, it will run the function in base class not the derived one. The solution for this issue is to add the keyword **virtual** before the function declaration in the derived class. This will make the pointer call the function in the derived.

```
class base{
   public: virtual void print() {cout << "Base";}
};
class derived : public base {
   public: void print() {cout << "Derived";}
};
void main{
   derived D1;
   base *ptr = &D1;
   ptr->print();
}
```

Besides, the pure virtual function is the one where it has no data in the base class and only get one in the derived. Also, this type of base class is called abstract class and it has no constructors. However, it is totally different from the normal virtual.

```
class base{
   public:
      virtual char* getName() const = 0;
};

class derived : public base {
   public:
      char* getName() {
         return "derived";
      }
};
```

UML Cheatsheet

UML is a mechanism for communication. It is intended to convey the meaningful parts of your application. Include the data which will help someone understand your code, not everything must be included (unless it's an exam, then include everything).

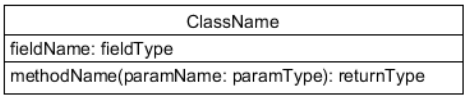
Representing Classes

The basic method for representing fields and methods is:

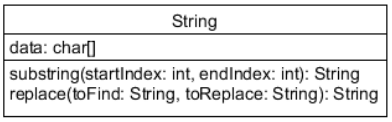
Fields:	**Methods:**
name: Type	name(paramName1: Type, paramName2: Type): returnType

Below is a general template for representing classes, and a small representation of the String class. If you're representing an interface, put <<interface>> above the class name, for an abstract class, put the name in *italics*.

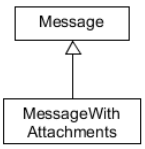
Template:



String Representation:

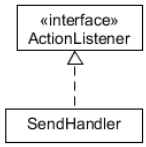


Arrows:



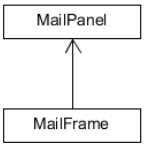
Inheritance
(is-a)

MessageWithAttachment is-a Message

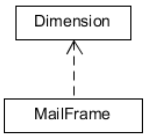


Interface Implementation
(is-a)

SendHandler is-an ActionListener

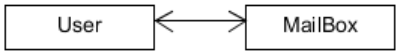


Association
(has-a field,
or has a field that uses the type as a type parameter)
MailFrame has-a MailPanel field

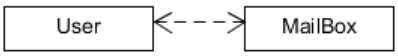


Dependency
(depends on, or uses but is not a field)

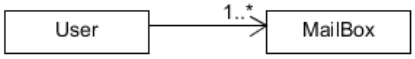
MailFrame depends-on Dimension



Two-Way Association
User has a MailBox field AND
MailBox has a User field



Two-Way Dependency
User depends on MailBox AND
MailBox depends on User
(ex. A MailBox method takes a User as a parameter, or vice versa)



Cardinality
How many items are in a relationship
Say: "A User has one-to-many MailBoxes"