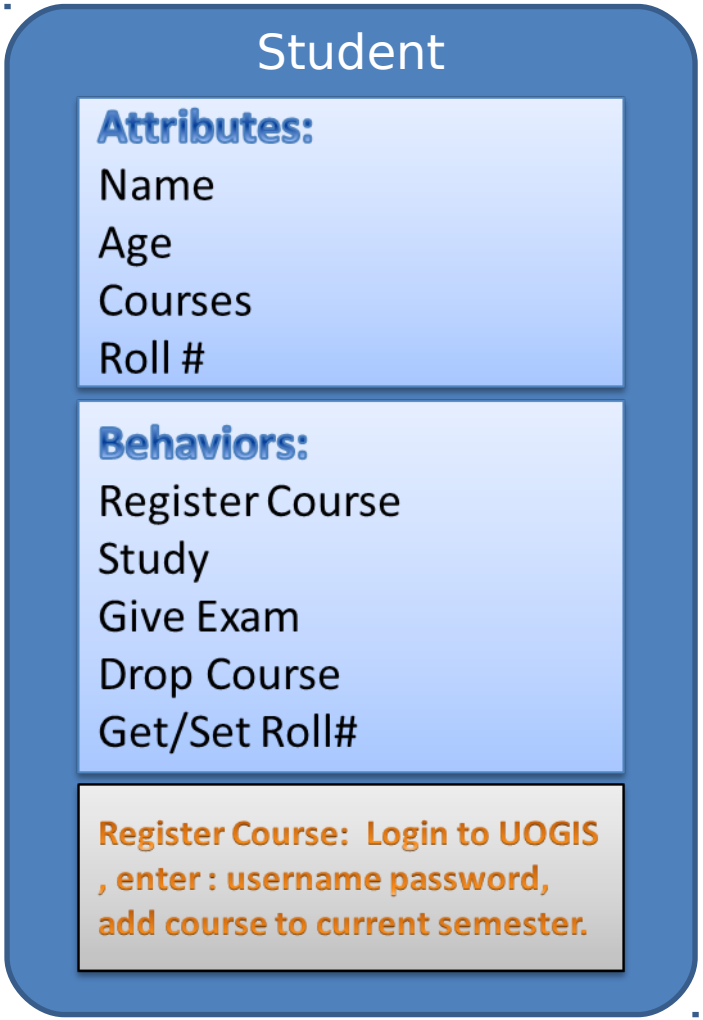
Object Oriented Programming

Using C++ Programming Language

RECAP

- Information hiding
- Encapsulation
- Implementation
- Interface
- Messages
- Abstraction



Lecture # 4

Functions, Function arguments, Function overloading, Classes and Objects

Functions

A programming unit that perform a particular task.

“A function groups a number of statements into a unit and gives it a name known as function name”

```
void ShowWelcomeMessage()
{
    cout<<“Welcome to OOP”;
}
```

Functions…

Function Declaration

```
void ShowWelcomeMessage();
```

Function Calling

```
ShowWelcomeMessage();
```

Function Definition

```
void ShowWelcomeMessage()
{
    cout<<“Welcome to OOP”;
}
```

Why needs separate Declaration and Definition? Can we eliminate

Passing arguments

Passing by Value

```
int Add(int A, int B){
      return A+B;
}
```

```
Add( 3 , 4);
```

```
Add(nVal1, nVal2);
```

Passing by Reference

```
bool IsHorizontalLine(Line* line)
{
     if(line->PointA.y-cord == line->PointB.y-cord)
     return TRUE;
}
```

Function Overloading

Function declaration is also known as prototype or signature;

```
void ShowWelcomeMessage();
```

```
void ShowWelcomeMessage(char *Msg);
```

```
void ShowWelcomeMessage(char *Msg, int len);
```

```
void ShowWelcomeMessage(char *Msg, COLORREF clr, int len);
```

```
void ShowWelcomeMessage()
{
    ShowWelcomeMessage(“Welcome to UOG…”);
}
```

Inline Functions

We can specify function arguments default values

```
void ShowWelcomeMessage(char *Msg, int len, COLORREF clr=0xFF0000);
```

Inline function provides fast execution by avoid branch taken.

```
Inline void ShowWelcomeMessage()
{
    ShowWelcomeMessage(“Welcome to UOG…”);
}
```

By Reference vs by pointer

- A pointer can be re-assigned any number of times while a reference can not be reassigned after initialization.
- A pointer can point to NULL while reference can never point to NULL
- You can't take the address of a reference like you can with pointers
- There's no "reference arithmetics" (but you can take the address of an object pointed by a reference and do pointer arithmetics on it as in &obj + 5).

To clarify a misconception:

Almost every C++ compiler implements references as pointers. That is, a declaration such as:

int &ri = i;
allocates the same amount of storage as a pointer, and places the address of i into that storage.

So pointer and reference occupies same amount of memory

As a general rule,

• Use references in function parameters and return types to define attractive interfaces.

• Use pointers to implement algorithms and data structures.

Therefore References are syntactic sugar, so easier code to read and write :)

Function Default Arguments

We can specify function arguments default values

```
void ShowWelcomeMsg(char *Msg, int len = 0, COLORREF clr=0xFF0000);
```

```
void ShowWelcomeMsg(char *Msg, int len, COLORREF clr)
{
    SetForeGroundColor(clr);
    cout<<GetSubString(Msg,len);
}
```

```
void ShowWelcomeMessage("Welcome to UOG...");
void ShowWelcomeMessage ("Welcome to UOG...",17);
void ShowWelcomeMessage ("Welcome to UOG...",8,0x00FF00);
```

External Variables and Lifetime

External variables exists for the life of the program.

Function level scope

```
int RollNumber = 999;
void DisplayRollNumber()
{
    int RollNumber = 345;
    cout<<RollNumber
}
```

Static variables

```
void IncrementCounter()
{
    static int count= 0;
    cout<<count;
}
```

Static variable have program level life time.

Constant function arguments

const float PI = 3.14;

```
int Add(const int A, const int B)
{
    // A = 5;    Results  Error
    return A+B;
}
```

Constant function arguments

const float PI = 3.14;

```
int Add(const int& A, const int& B)
{
    // A = 5;    Results  Error
    return A+B;
}
```

Q & A