Lecture - 11

Proxy classes

Function overloading

Separating interface from implementation

- Define member functions outside the class definition, so that their implementation details can be hidden from the client code
- Divide the source code into two parts
  - A .cpp file having main() *(driver program)*
  - A .h file having class definition
  - A .cpp file having member function definition

employee.h

```
#include <iostream>
#include <cstring>
using namespace std;
class employee
{  // class begins
char name[80];
public: void putname(char *); void getname(char *) ;
private: double wage;
public: void putwage(double w); double getwage();
}; // class ends here
```

employee.cpp

```
#include “employee.h”
void employee::putname(char *n)
{ strcpy(name,n); }
void employee::getname(char *n)
{ strcpy(n,name); }
void employee::putwage(double w)
{ wage=w;}
double employee::getwage() { return wage; }
```

program1.cpp

```
#include <iostream>
#include “employee.h”
int main()
{ employee ted; char
    name[80];
ted.putname(“Ted
    Jones”);
    ted.putwage(7500);
ted.getname(name);
cout<<name<<“ make
    $”<<ted.getwage()<
    <“ per month.” ;


return 0;
} // main closing
```

Problem

- In this scenario, complete information hiding do not occur as class's private data is exposed in .h file

- To avoid this – one can use the concept of ***proxy classes***

Proxy classes

- Allows us to hide even the private data of a class from clients of the class
- Proxy classes are classes which provide interface to the original class whose implementation details need to be hidden

Example

Step 1. Make the class in header file "implementation.h"

```
class implementation
{ public:
   implementation(int v)
   { value=v;}
void setvalue(int v) {
   value=v;}
int getvalue() { return
   value;}
private: int value;
};
```

Step 2. Make the proxy class in another header file "interface.h"

```
#include "implementation.h"
class interface
{ public:
interface(int);
void setalue(int);
int getvalue();
~interface();
private:
Implementation *ptr;
};
```

Contd..

Step 3. Provide implementation of interface class in "interface.cpp"

```
#include "implementation.h"
#include "interface.h"
interface::interface(int v)
{ ptr=new implementation(v) ; }
void interface::setvalue(int v)
{ ptr->setvalue(v); }
int interface::getvalue()
{ return ptr->getvalue(); }
Interface::~interface()
{ delete ptr };
```

Step 4. Main program (client of class) or driver program "user.cpp"

```
#include "interface.h"
void main()
{
interface k(5); int j ;
j=k.getvalue();
cout<<j;
k.setvalue(10);
j=k.getvalue();
cout<<j;
}
```

Polymorphism

- Polymorphism (*poly* = many and *morph* = states or forms, etc.)

- A function is polymorphic if it may be applied to arguments of different types.

```
void add(int a,int b);
void add(float c,float d);
```

Polymorphism and overloading

- Realized by using a set of mono-morphic functions. Different code is used for different types.

- Overloading means that the same operation is implemented through different methods with the same function name. For different types, different implementations (methods) of the same operation are executed.

Example

```
#include<iostream>
void add(int a,int b)
{ int c; c=a+b;
  cout<<c; }
void add(float c,float
  d)
{ float x; x=a+b;
  cout<<c; }
```

```
void main()
{
 add(5,10);
add(15.2,12.6);
}
```

Class assignment

- Write a program that uses a function min to determine the smaller of two arguments. Test the program using int, char and float arguments.