CSE 107: Object Oriented Programming Language

Dr. Tanzima Hashem
Professor
CSE, BUET

CLASSES

- C++ Classes are the logical abstraction or model of C++ Objects
- A Class declaration defines a new type
- It determines what an object of that type will look like
- It determines the nature of the data and functions of that type
- Classes must be defined before creating the objects, i.e., objects cannot be created without the classes
- Definition of a class does not create any physical objects rather a logical abstraction

CLASSES

- General syntax -

```
class class-name
{
        // private functions and variables
public:
        // public functions and variables
}object-list (optional);
```

Classes

- Member access specifiers

- public:
  - can be accessed outside the class directly
    - The public stuff is *the interface*

- private:
  - Accessible only to member functions of class
  - Private members and methods are for internal use only

Classes

```
class Rectangle{
        int x, y;
public:
        void set_values (int,int);
        int area() {return (x*y);}
};

void Rectangle::set_values (int a, int b) {
        x = a;
        y = b;
}
```

CLASSES

```
class Circle{
        double radius;
public:
        void setRadius(double r) {radius = r;}
        double getDiameter() { return radius *2;}
        double getArea();
        double getCircumference();
};
double Circle::getArea()
{
    return radius * radius * (22.0/7);
}
double Circle:: getCircumference()
{
    return 2 * radius  * (22.0/7);
}
```

Classes

- General syntax -

```
class class-name
{
        // private functions and variables
public:

        // public functions and variables
}object-list (optional);
```

Classes

- Member access specifiers

- public:
  - Can be accessed outside the class directly
    - The public stuff is *the interface*

- private:
  - Accessible only to member functions of class
  - Private members and methods are for internal use only

Classes

```
class Rectangle{
        int height, width;
public:
        void set_values (int,int);
        int area() {return (height*width);}
};

void Rectangle::set_values (int a, int b) {
        height = a;
        width = b;
}
```

Classes

```
class Circle{
        double radius;
public:
        void setRadius(double r) {radius = r;}
        double getDiameter() { return radius *2;}
        double getArea();
        double getCircumference();
};
double Circle::getArea()
{
    return radius * radius * (22.0/7);
}
double Circle:: getCircumference()
{
    return 2 * radius  * (22.0/7);
}
```

C++ OBJECTS

- C++ Classes are used as the type specifier to create C++ Objects

  Rectangle recta, rectb;

- An object declaration creates a physical entity of its class type, i.e., occupies memory space class type

- Each object has its own copy of data

C++ Objects

```
int main () {

    Rectangle recta, rectb;
    recta.set_values (3,4);
    rectb.set_values (5,6);

    cout << "recta area: " << recta.area() << endl;
    cout << "rectb area: " << rectb.area() << endl;

    recta.height=5;

    // Not possible, height is a private member

    return 0;
}
```

USING NEW AND DELETE

- C++ introduces two operators for dynamically allocating and deallocating memory

- **p_var = new type   //type *p_var;**
  - new returns a pointer to dynamically allocated memory that is sufficient to hold a data of type
- **delete p_var**
  - releases the memory previously allocated by new

- Memory allocated by new must be released using delete

USING NEW AND DELETE

- In case of insufficient memory, ***new*** can report failure in two ways
  - By returning a null pointer
  - By generating an exception
- The reaction of ***new*** in this case varies from compiler to compiler

USING NEW AND DELETE

- Advantages
  - Automatically allocates enough memory to hold an object of the specified type, do not need to use sizeof operator
  - Automatically returns a pointer of the specified type, do not to use an explicit type cast
  - Both new and delete can be overloaded
  - In case of objects, new dynamically allocates the object and calls its constructor
  - In case of objects, delete calls the destructor of the object

CONSTRUCTOR

- Special member function
  - Public function member
  - Same name as class
  - No return type

- Indicate parameters in prototype:

```
Rectangle(double, double);
```

- Use parameters in the definition:

```
Rectangle::Rectangle(int a, int b){
        height = a;
        width = b;
}
```

- Declare objects with parameters

```
Rectangle r(10, 5);
```

Constructor

- Automatically called when a new object is created (instantiated)
- Initialize data members
- Several constructors
  - Function overloading

```
Rectangle();

Rectangle(int);

Rectangle(int, int);
```

Default Constructor

- A constructor function with no parameter

  Rectangle();

- Supplied by the compiler automatically if no constructor is defined by the programmer
  - Does not initialize the member variables to any default value
  - Contain garbage value after creation

CONSTRUCTOR

```
class Circle{
        double radius;
public:
        Circle() { radius = 0.0;}
        Circle(int r);
        void setRadius(double r) {radius = r;}
        double getDiameter() {return radius *2;}
        double getArea();
        double getCircumference();
};
```

Constructor

```
Circle::Circle(int r)
{
    radius = r;
}
double Circle::getArea()
{
    return radius * radius * (22.0/7);
}
double Circle:: getCircumference()
{
    return 2 * radius  * (22.0/7);
}
```

Constructor

```
int main()
{
   Circle c1, c2(7);

   c1.setRadius(5);

   cout<<“The area of c1:”<<c1.getArea()<<“\n”;

   cout<<“The circumference of c1:”<< c1.getCircumference()<<“\n”;

   cout<<“The Diameter of c2:”<<c2.getDiameter()<<“\n”;

   return 0;
}
```

Destructor

- Special member function
  - Public function member
  - Same name as class
  - Preceded with tilde (~),
    - e.g., **~Rectangle() { cout << "Destructor"; }**
  - No arguments
  - No return value
- Automatically called by the compiler when an object is destroyed
- Mainly used to de-allocate dynamic memory locations
- Cannot be overloaded

Constructor and Destructor

```
#include <iostream>
using namespace std;

class Rectangle {
        int *width, *height;
public:
        Rectangle(int, int);
        ~Rectangle ();
        int area () {return (*width * *height);}
};

Rectangle::Rectangle (int a, int b) {
        width= new int;
        height = new int;
        *width = a;
        *height = b;
}
```

Constructor and Destructor

```
Rectangle:: ~Rectangle () {
        delete width;
        delete height;
}

int main () {
        Rectangle recta (3,4), rectb (5,6);
        cout << "recta area: " << recta.area() << endl;
        cout << "rectb area: " << rectb.area() << endl;
        return 0;
}
```

Constructor and Destructor

- For global objects, an object's constructor is called once, when the program first begins execution

- For local objects, the constructor is called each time the declaration statement is executed

- Local objects are destroyed when they go out of scope

- Global objects are destroyed when the program ends

Object Pointer

- It is possible to access a member of an object via a pointer to that object
- Creation of an object pointer does not create an object
- We can take the address of objects using the address operator (&) and store it in object pointers
  - **A ob; A *p = &ob;**
- We have to use the arrow (->) operator instead of the dot (.) operator while accessing a member through an object pointer
  - **p->f1(); *// let f1 is public in A***
- Pointer arithmetic using an object pointer is the same as it is for any other data type
  - When incremented, it points to the next object
  - When decremented, it points to the previous object

Object Pointer

```
int main () {
    Rectangle a, *b, *c;

    b= new Rectangle;
    c= &a;

    a.set_values (1,2);
    b->set_values (3,4);

    cout << "a area: " << a.area() << endl;
    cout << "*b area: " << b->area() << endl;
    cout << "*c area: " << c->area() << endl;

    delete b;
    return 0;
}
```

ASSIGNING OBJECTS

- One object can be assigned to another provided that both objects are of the same type
- It is not sufficient that the types just be physically similar – their type names must be the same
- By default, when one object is assigned to another, a bitwise copy of all the data members is made, including compound data structures like arrays
- Creates problem when member variables point to dynamically allocated memory and destructors are used to free that memory

Assigning Objects

```
#include <iostream>
using namespace std;

class Rectangle {
    int width, height;
public:
    Rectangle (int,int);
    int area (){return(width*height);}
};

Rectangle::Rectangle (int a, int b) {
        width = a; height = b;
}
```

```
int main () {
    Rectangle recta (3,4);
    Rectangle rectb (5,6);

    rectb=recta;

    cout << "recta area: " << recta.area()
    << endl;
    cout << "rectb area: " << rectb.area()
    << endl;

    return 0;
}
```

Assigning Objects

```
#include <iostream>
using namespace std;

class Rectangle {
        int *width, *height;
public:
        Rectangle(int, int);
        ~Rectangle ();
        int area () {return (*width * *height);}
};

Rectangle::Rectangle (int a, int b) {
        width= new int;
        height = new int;
        *width = a;
        *height = b;
}
```

Assigning Objects

```
Rectangle:: ~Rectangle () {
        delete width;
        delete height;
}

int main () {
        Rectangle recta (3,4), rectb (5,6);
        recta=rectb;
        cout << "recta area: " << recta.area() << endl;
        cout << "rectb area: " << rectb.area() << endl;
        return 0;
}
```

PASSING OBJECTS TO FUNCTIONS

- Objects can be passed to functions as arguments in just the same way that other types of data are passed

- By default all objects are passed by value to a function
- Address of an object can be sent to a function to implement call by reference

- In call by reference, as no new objects are formed, constructors and destructors are not called
- But in call by value, while making a copy, constructors are not called for the copy but destructors are called

Returning Objects from Functions

- The function must be declared as returning a class type
- When an object is returned by a function, a temporary object (invisible to us) is automatically created which holds the return value
- While making a copy, constructors are not called for the copy but destructors are called
- After the value has been returned, this object is destroyed
- The destruction of this temporary object might cause unexpected side effects in some situations

Passing Objects to Functions/Returning Objects from Functions

```
#include <iostream>
using namespace std;

class Rectangle {
        int width, height;
public:
        Rectangle (int,int);
        int area () {return (width*height);}
};

Rectangle::Rectangle (int a, int b) {
        width = a; height = b;
}
```

Passing Objects to Functions/Returning Objects from Functions

```
Rectangle larger(Rectangle recta, Rectangle rectb){
        if(recta.area()>rectb.area())
                return recta;
        else
                return rectb;
}

int main () {
        Rectangle recta (3,4);
        Rectangle rectb (5,6);
        Rectangle rect_larger(0,0);

        rect_larger=larger(recta, rectb);

        cout << "recta area: " << recta.area() << endl;
        cout << "rectb area: " << rectb.area() << endl;
        cout << "rect_larger area: " << rect_larger.area() << endl;
        return 0;
}
```

Passing Objects to Functions/Returning Objects from Functions

```
#include <iostream>
using namespace std;

class Rectangle {
        int *width, *height;
public:
        Rectangle(int, int);
        ~Rectangle ();
        int area () {return (*width * *height);}
};

Rectangle::Rectangle (int a, int b) {
        width= new int;
        height = new int;
        *width = a;
        *height = b;
}
```

Passing Objects to Functions/Returning Objects from Functions

```
Rectangle:: ~Rectangle () {
        delete width;
        delete height;
}

Rectangle larger(Rectangle recta, Rectangle rectb){
        if(recta.area()>rectb.area())
                return recta;
        else
                return rectb;
}
```

Passing Objects to Functions/Returning Objects from Functions

```
int main () {
        Rectangle recta (3,4);
        Rectangle rectb (5,6);
        Rectangle rect_larger(0,0);

        rect_larger=larger(recta, rectb); //this will cause the program to crash

        cout << "recta area: " << recta.area() << endl;
        cout << "rectb area: " << rectb.area() << endl;
        cout << "rect_larger area: " << rect_larger.area() << endl;
        return 0;
}
```

In-line Functions

- Functions that are not actually called but, rather, are expanded in line, at the point of each call

- Advantage
  - Have no overhead associated with the function call and return mechanism
    - Can be executed much faster than normal functions
  - Safer than parameterized macros

- Disadvantage
  - If they are too large and called too often, the program grows larger

In-line Functions

```
inline int even(int x)
{
        return !(x%2);
}

int main()
{
        if(even(10)) cout << “10 is even\n”;
        // becomes if(!(10%2))

        if(even(11)) cout << “11 is even\n”;
        // becomes if(!(11%2))

        return 0;
}
```

In-line Functions

- The **inline** specifier is a *request*, not a command, to the compiler

- Some compilers will not in-line a function if it contains
  - A **static** variable
  - A **loop**, **switch** or **goto**
  - A **return** statement
  - If the function is **recursive**

AUTOMATIC IN-LINING

- Defining a member function inside the class declaration causes the function to automatically become an in-line function
- In this case, the **inline** keyword is no longer necessary
  - However, it is not an error to use it in this situation
- Same restrictions that apply to “normal” in-line function apply to automatic in-line function

Automatic In-lining

```
// Automatic in-lining
class myclass
{
        int a;
public:
        myclass(int n) { a = n; }
        void set_a(int n) { a = n; }
        int get_a() { return a; }
};
```

```
// Manual in-lining
class myclass
{
        int a;
public:
        myclass(int n);
        void set_a(int n);
        int get_a();
};
inline void myclass::set_a(int n)
{
        a = n;
}
```

In-line Functions

```
#include <iostream>

inline void increment(int n) { n = n + 1; }`

int main() {
   int n = 0;
   increment(n);
   std::cout << "Result " << n;
}
```

In-line Functions

```
#include<iostream>
using namespace std;

inline void set(int x, int y){
   x = y;
}

int main(){
   int a = 3, b = 5;
   cout << a << " " << b << endl;
   set(a,b); //I would think this is replaced by a = b;
   cout << a << " " << b << endl;
   return 0;
}
```

Friend Functions

- A friend function is not a member of a class but still has access to its private elements
- A friend function can be
  - A global function not related to any particular class
  - A member function of another class
- Inside the class declaration for which it will be a friend, its prototype is included, prefaced with the keyword **friend**
- Why friend functions ?
  - Operator overloading
  - Certain types of I/O operations
  - Permitting one function to have access to the private members of two or more different classes

Friend Functions

```
class MyClass
{
  int a; // private member
public:
  MyClass(int a1) {
    a = a1;
  }
  friend void ff1(MyClass obj);
};
```

```
// friend keyword not used
void ff1(MyClass obj)
{
  cout << obj.a << endl;
  MyClass obj2(100);
  cout << obj2.a << endl;
}
int main()
{
  MyClass o1(10);
  ff1(o1);
  return 0;
}
```

FRIEND FUNCTIONS

- A friend function is not a member of the class for which it is a friend
  - **MyClass obj(10), obj2(20);**
  - **obj.ff1(obj2); // wrong, compiler error**

- Friend functions need to access the members (private, public or protected) of a class through an object of that class
  - The object can be declared within or passed to the friend function

- A member function can directly access class members

- A function can be a member of one class and a friend of another

Friend Functions

```
class YourClass; // a forward declaration

class MyClass {
  int a; // private member
public:
  MyClass(int a1) { a = a1; }
  friend int compare (MyClass obj1,
   YourClass obj2);
};

class YourClass {
  int a; // private member
public:
  YourClass(int a1) { a = a1; }
  friend int compare (MyClass obj1,
   YourClass obj2);
};

int main() {
  MyClass o1(10); YourClass o2(5);
  int n = compare(o1, o2); // n = 5
  return 0;
}

int compare (MyClass obj1, YourClass obj2)
{
  return (obj1.a – obj2.a);
}
```

Friend Functions //Making a function of other class a friend

```
class YourClass; // a forward declaration

class MyClass {
  int a; // private member
public:
  MyClass(int a1) { a = a1; }
  int compare (YourClass obj);
};

class YourClass {
  int a; // private member
public:
  YourClass(int a1) { a = a1; }
  friend int MyClass::compare (YourClass
 obj);
};
```

```
int MyClass::compare (YourClass obj) {
    return (a - obj.a);
  }

int main() {
  MyClass o1(10); Yourclass o2(5);
  int n = o1.compare(o2); // n = 5
  return 0;
}
```

ARRAYS OF OBJECTS

- Arrays of objects of class can be declared just like other variables
  - **class A{ … };**
  - **A ob[4];**
  - **ob[0].f1();** ***// let f1 is public in A***
  - **ob[3].x = 3;** ***// let x is public in A***

- In this example, all the objects of the array are initialized using the default constructor of **A**

ARRAYS OF OBJECTS

- If a class type includes a constructor, an array of objects can be initialized
- Initializing array elements with the constructor taking an integer argument
  - ***class A{ public: int a; A(int n) { a = n; } };***
  - **A ob[2] = { A(-1), A(-2) };**
  - **A ob2[2][2] = { A(-1), A(-2), A(-3), A(-4) };**
- In this case, the following shorthand form can also be used
  - **A ob[2] = { -1, -2 };**

Arrays of Objects

- If a constructor takes two or more arguments, then only the longer form can be used

  - ***class A{ public: int a, b; A(int n, int m) { a = n; b = m; } };***

  - **A ob[2] = { A(1, 2), A(3, 4) };**

  - **Aob2[2][2] = { A(1, 1), A(2, 2), A(3, 3), A(4, 4) };**

ARRAYS OF OBJECTS

- We can also mix no argument, one argument and multi-argument constructor calls in a single array declaration.

```
class A
{
        public:
                A() { … } // must be present for this example to be compiled
                A(int n) { … }
                A(int n, int m) { … }
};
A ob[3] = { A(), A(1),A(2, 3) };
```

ARRAYS OF OBJECTS (PRACTICE)

```
#include <iostream>
using namespace std;
class Circle{
            double radius;
            double x;
            double y;
public:
Circle(double r){radius=r; x=0; y=0;}
Circle(double r, double c1, double c2)
{radius=r; x=c1; y=c2;}

double area(){return 3.14*radius*radius;}
}
```

```
int main(){

Circle crls1[2]={7.44, 3.65};

Circle crls2[2]={Circle(7.44, 0.0, 0.0),
  Circle(3.65, 0.5, 2.5)};

return 0;
}
```

THIS POINTER

- A special pointer in C++ that points to the object that generates the call to the method

- The compiler automatically adds a parameter whose type is "pointer to an object of the class" in every non-static member function of the class
  - ***class A{ public: void f1() { … } };***
  - ***class A{ public: void f1(A *this) { … } };***

- It also automatically calls the member function with the address of the object through which the function is invoked
  - **A ob; ob.f1();**
  - **A ob; ob.f1(&ob);**

THIS POINTER

- It is through this pointer that every non-static member function knows which object's members should be used

```
class A
{
	int x;
public:
	void  f1()
	{
		x = 0; // this->x = 0;
	}
};
```

THIS POINTER

- this pointer is generally used to access member variables that have been hidden by local variables having the same name inside a member function

```
class A{
        int x;
   public:
     A(int x) {
        x = x; // only copies local 'x' to itself; the member 'x' remains uninitialized
        this->x = x; // now its ok
     }
};
```

MORE ABOUT NEW AND DELETE

- Dynamically allocated objects can be given initial values

  - **int *p = new int;**
    - Dynamically allocates memory to store an integer value which contains garbage value

  - **int *p = new int(10);**
    - Dynamically allocates memory to store an integer value and initializes that memory to 10
    - *Note the use of parenthesis **( )** while supplying initial values*

More About new and delete

- **class A{ int x; public: A(int n) { x = n; } };**
  - **A *p = new A(10);**
    - Dynamically allocates memory to store a A object and calls the constructor A(int n) for this object which initializes x to 10
  - **A *p = new A;**
    - It will produce **compiler error** because in this example class A does not have a default constructor

More About new and delete

- We can also create dynamically allocated arrays using new
- But deleting a dynamically allocated array needs a slight change in the use of delete

- It is not possible to initialize an array that is dynamically allocated

  - **int *a= new int[10];**
    - Creates an array of 10 integers
    - All integers contain garbage values
    - *Note the use of square brackets [ ]*

  - **delete [ ] a;**
    - Delete the entire array pointed by a
    - *Note the use of square brackets [ ]*

More About new and delete

- We can also create dynamically allocated arrays using new
- But deleting a dynamically allocated array needs a slight change in the use of delete

- It is not possible to initialize an array that is dynamically allocated

  - **int *a= new int[10];**
    - Creates an array of 10 integers
    - All integers contain garbage values
    - *Note the use of square brackets [ ]*

  - **delete [ ] a;**
    - Delete the entire array pointed by a
    - *Note the use of square brackets [ ]*

More About new and delete

- It is not possible to initialize an array that is dynamically allocated
- In order to create an array of objects of a class, the class must have a default constructor

```
class A {
   int x;
public:
   A(int n) { x = n; } };

A *array = new A[10];
// compiler error
```

```
class A {
   int x;
public:
   A() { x = 0; }
   A(int n) { x = n; } };
A *array = new A[10]; // no error
// use array
delete [ ] array;
```

MORE ABOUT NEW AND DELETE

- **A *array = new A[10];**
  - The default constructor is called for all the objects.
- **delete [ ] array;**
  - Destructor is called for all the objects present in the array.

REFERENCES

- A reference is an implicit pointer
- Acts like another name for a variable
- Can be used in three ways
  - A reference can be passed to a function
  - A reference can be returned by a function
  - An independent reference can be created
- Reference variables are declared using the & symbol
  - **void f(int &n);**
- Unlike pointers, once a reference becomes associated with a variable, it cannot refer to other variables

References

- **Using pointer -**

```
void f(int *n) {
   *n = 100;
}
int main() {
   int i = 0;
   f(&i);
   cout << i; // 100
   return 0;
}
```

- **Using reference -**

```
void f(int &n) {
    n = 100;
}
int main() {
   int i = 0;
   f(i);
   cout << i; // 100
   return 0;
}
```

References

- A reference parameter fully automates the call-by-reference parameter passing mechanism

  - No need to use the address operator (&) while calling a function taking reference parameter

  - Inside a function that takes a reference parameter, the passed variable can be accessed without using the indirection operator (*)

REFERENCES (PRACTICE)

```
#include <iostream>
using namespace std;
void swapargs (int x, int y){
    int t;
    t=x;x=y;y=t;
}
int main(){
    int i, j;
    i=20; j=40;
    cout<<"i="<<i<<", "<<j<<endl;
    swapargs(i,j);
    cout<<"i="<<i<<", "<<j<<endl;
    return 0;
}
```

```
#include <iostream>
using namespace std;
void swapargs (int *x, int *y){
    int t;
    t=*x;*x=*y;*y=t;
}
int main(){
    int i, j;
    i=20; j=40;
    cout<<"i="<<i<<", "<<j<<endl;
    swapargs(&i,&j);
    cout<<"i="<<i<<", "<<j<<endl;
    return 0;
}
```

References (Practice)

```
#include <iostream>
using namespace std;
void swapargs (int *x, int *y){
   int t;
   t=*x;*x=*y;*y=t;
}
int main(){
   int i, j;
   i=20; j=40;
   cout<<"i="<<i<<", "<<j<<endl;
   swapargs(&i,&j);
   cout<<"i="<<i<<", "<<j<<endl;
   return 0;
}
```

```
#include <iostream>
using namespace std;
void swapargs (int &x, int &y){
        int t;
        t=x;x=y;y=t;
}
int main(){
        int i, j;
        i=20; j=40;
        cout<<"i="<<i<<", "<<j<<endl;
        swapargs(i,j);
        cout<<"i="<<i<<", "<<j<<endl;
        return 0;
}
```

Passing References to Objects

- We can pass objects to functions using references
- No copy is made, destructor is not called when the function ends
- As reference is not a pointer, we use the dot operator (.) to access members through an object reference

Passing References to Objects

```
class myclass {
         int x;
public:
        myclass() {
                x = 0;
                cout << “Constructing\n”;
        }
        ~myclass() {
                cout << “Destructing\n”;
        }
        void setx(int n) { x = n; }
        int getx() { return x; }
};
```

```
void f(myclass &o) {
        o.setx(500);
}
int main() {
  myclass obj;
  cout << obj.getx() << endl;
  f(obj);
  cout << obj.getx() << endl;
  return 0;
}
```

Output:
Constructing
0
500
Destructing

Returning References

- A function can return a reference
- Allows a functions to be used on the left side of an assignment statement
- But, the object or variable whose reference is returned must not go out of scope
- So, we should not return the reference of a local variable
  - For the same reason, it is not a good practice to return the pointer (address) of a local variable from a function

Returning References

```
int x; // global variable
int &f() {
  return x;
}

int main() {
  x = 1;
  cout << x << endl;
  f() = 100;
  cout << x << endl;
  x = 2;
  cout << f() << endl;
  return 0;
}
```

Output:

```
1
100
2
```

REFERENCES

- **Advantages**
  - The address is automatically passed
  - Reduces use of ‘&’ and ‘*’
  - When objects are passed to functions using references, no copy is made
    - Hence destructors are not called when the functions ends
    - Eliminates the troubles associated with multiple destructor calls for the same object

Independent References

- Simply another name for another variable
- Must be initialized when it is declared
  - **int &ref;** *// compiler error*
  - **int x = 5; int &ref = x;** *// ok*
  - **ref = 100;**
  - **cout << x;** *// prints "100"*

- An independent reference can refer to a constant
  - **int &ref=10;** *// compile error*
  - **const int &ref = 10;**

Restrictions

- We cannot reference another reference
  - Doing so just becomes a reference of the original variable
- We cannot obtain the address of a reference
  - Doing so returns the address of the original variable
  - Memory allocated for references are hidden from the programmer by the compiler
- We cannot create arrays of references
- We cannot reference a bit-field
- References must be initialized unless they are members of a class, are return values, or are function parameters

Acknowledgement
http://faizulbari.buet.ac.bd/Courses.html
http://mhkabir.buet.ac.bd/cse201/index.html

THE END

Topic Covered: Chapters 2 (2.1, 2.2, 24, 2.6, 2.7), 3, 4