# Unit 2
# Object-Oriented Programming with C++

- *Overview of Object-Oriented Programming*
- *C++ Classes*
- *Constructors*
- *Pointers and Dynamic Objects*
- *Destructors*
- *Overloading Operators*
- *Conversion Operators*
- *Memory Organization*
- *Dynamic Memory Allocation*
- *Input/Output*

# Object-Oriented Programming (OOP)

- Programs are collections of objects
- Computation is done by sending messages to objects
- Objects have 3 characteristics
  - state
  - behavior
  - identity
- **Example**: bank accounts *state*: id, name, balance
  - *behaviour*: deposit, withdraw, etc.
  - *identity*: Joe's account is similar but different than Jane's
- A *class* is a group of objects with the same behavior and representation.
  - A class defines an abstract data type
    - it defines the state variables and operations (methods) for its objects.
  - A class may inherit characteristics from ancestor classes.
  - Objects are instances of classes.
- Fundamental characteristics of OOP languages are:
  - *encapsulation*
  - *instantiation*
  - *inheritance*
  - *polymorphism*

# C++ Classes

- A *class declaration* or *signature* consists of:
  - class name
  - declarations of private members (data and operations )
  - declarations of public members (data and operations)
- A *class definition* consists of:
  - a class declaration, and
  - definitions of the member functions (operations)
- It is customary to treat a class as a module, splitting its definition is a **specification** and **implementation files**
- The *specification* or *header* file of a class X
  - contains the class declaration
  - is named X.h
  
  .
- The implementation or source file of a class X
  - contains the definitions of the member functions
  - is,named X.cpp or X.C .
- Any file that uses X can include X.h.



# Example: Class Point, Header File

```
// file: Point.h
// Point class header

#ifndef POINT_H
#define POINT_H

class Point

   // The Point class defines a point on the
      Cartesian plane in terms of x and
   // y coordinates. The class has both a default
      and parameterized
   // constructors.

   // Class Invariants:
   // - The coordinates of a Point are always
      defined
   // - getX() and getY() always return a value
{
  public:

   Point();
   // Default constructor
   // PRE:  None
   // POST: A new Point object is created with (
      0, 0 ) as its coordinates

Point( double x1, double y1 );
  // Parameterized constructor
  // PRE:  x1 and y1 are two valid doubles.
  // POST: A new Point object is created with (
      x1, y1 ) as its coordinates

  double getX() const;
  // Accessor member function
  // PRE:  None
  // POST: The current X coordinate is returned


double getY() const;
  // Accessor member function
  // PRE: None
  // POST: The current Y coordinate is returned

  double distanceFromOrigin() const;
  // Accessor member function
  // PRE: None
  // POST: The distance to the origin is returned
  // Library facilities used: sqrt() from cmath


#endif
```

# Class Point, Header File (cont')

```
Point translate( double xDistance, double
    yDistance ) const;
  // Accessor member function
  // PRE:  xDistance and yDistance are the
     horizontal and vertical
  //     displacements.
  // POST: A Point located at the result of the
     translation is returned

  void print() const;
  // Accessor member function
  // PRE:  None
  // POST: The current coordinates of the Point
     is printed
  // Library facilities used: cout object and
     operator<< from iostream

 private:

  double x;   // x coordinate
  double y;   // y coordinate

};
```



# Class Point, Source File

```
// file: Point.cpp
// Point class source

#include <iostream>    // used in print()
#include <cmath>       // used in
     distanceFromOrigin()
#include "Point.h"
using namespace std;

// Note: This module uses the sqrt() function in
     the math library.
//    So when producing a final executable file
     using this module,
//    the math library must be linked in.

Point::Point()
// Default constructor
// PRE:  None
// POST: A new Point object is created with ( 0,
     0 ) as its coordinates
{
  x = 0;
  y = 0;
}

Point::Point( double x1, double y1 )
// Parameterized constructor
// PRE:  x1 and y1 are two valid doubles.
// POST: A new Point object is created with ( x1,
     y1 ) as its coordinates
{
  x = x1;
  y = y1;
}


double Point::getX() const
// Accessor member function
// PRE:  None
// POST: The current X coordinate is returned
{
  return x;
}
```

# Class Point, Source File (cont'd)

```
double Point::getY() const
// Accessor member function
// PRE: None
// POST: The current Y coordinate is returned
{
  return y;
}

double Point::distanceFromOrigin() const
// Accessor member function
// PRE: None
// POST: The distance to the origin is returned
// Library facilities used: sqrt() from cmath
{
  return sqrt( x * x + y * y );
}

Point Point::translate( double xDistance, double
      yDistance ) const
// Accessor member function
// PRE:  xDistance and yDistance are the
      horizontal and vertical displacements.
// POST: A Point located at the result of the
      translation is returned
{
  double a;
  double b;

  a = x + xDistance;
  b = y + yDistance;

  return Point( a, b );
}

void Point::print() const
// Accessor member function
// PRE:  None
// POST: The current coordinates of the Point is
      printed
// Library facilities used: cout object and
      operator<< from iostream
{
  cout << "( " << x << ", " << y << " )";
}
```



# Using the Point class

- Now, Point can be used as a new type:

```
Point p;
Point q(2,3);

double a = 1;
a = p.x; // Error
p.x = a; // Error

a = p.getX() // a becomes 0
p.print() // prints: (0,0)
q.print() // prints: (2,3)

a = (p.translate(5,10)).getX() // a is 5
```

**Note:**

- Objects of a class are initialized by the class *constructors* or *initializers*.
- Private members are only accessible inside the class definition.
- Public operations (methods) are applied to an object
  - i.e. p.print
- This object is the *implicit argument* of the operation.
- A const at the end of the function header means the implicit argument is not modified.

# Constructors (or Initializers)

- Constructors are special operators used to *initialize* an object.
- Are not real functions:
  - have the same name as the class
  - are not called explicitly
  - do not have a return type
- A class may have multiple constructors (with different arguments)
- Constructors of a class X are invoked when
  - a variable of class X is declared i.e.
    Point p;
    Point q(2,3);
  - a temporary object is created; i.e.
    a = Point(3,4);
    return Point(2,3);
  - a function with a parameter X is called
    i.e. suppose distance is a function calculating the distance between two points
    a = distance(p, q)
  - a function returning X returns, i.e.
    p.translate(5,10);
  - a variable with a member X is initialized.



# Class Invariants

- Special assertions that are true by any member of the class
- They are denoted by *Class Invariants*:
  - in the design of the class, or
  - as a comment at the beginning of the header file for the class
- For any class:
  - a constructor
    ~ must satisfy the Class Invariants upon its return
  - a member function
    - assumes that Class Invariants are true at invocation
    - must ensure that Class Invariants are true upon its return
- **Example:** The design for the Point class should contain the following class invariant.
  - The coordinates of any point are always defined.
    OR
  - getx and gety always return a value

# Pointers and Dynamic Data

- A *pointer* is a variable whose value is the memory address of another data structure.
- A pointer declaration:

  <type>* <variable>
- E.g.

  int *ip, *iq;
  Point* pp;
- Note:
  - The type of pp is pointer to Point.
  - The declaration allocates space for pp, but not for any point.
- Functions can also return pointer values.
  i.e. int* f(int x)

  f returns an address of an integer.
- C and C++ permit a variety of operations on pointers including
  - dereference
  - pointer arithmetic



# The operators * and &

- Are used to manipulate pointers:
  - * is the *dereference* or *indirection* operator
    - used to dereference a pointer value.
  - & is the *address-of* operator
    - gives the address of an object.
- **Example**: Demonstrating the use of * and &.

```
int x, y;
int* ip;

x = 2;
y = 3;
ip = &x;        // ip points to x now
y = *ip;        // the value of y is now 2
*ip = 0;        // the value of x is now 0
(*ip)++;        // the value of x is now 1
```

- The address of an object cannot be changed.
  &x = &y is not permitted.
- & is used to pass the address of a parameter to a function.

# Notation we Use with Pointers

Consider the following declarations:

| Declaration | Address | Memory cells | Name |
|---|---|---|---|
| int x = 5; | 3000 | 5 | x |
| int y = 10 | 5000 | 10 | y |
| int * p; | 5000 | | p |

When we execute

p = &x;

3000 is placed in the memory slot for p

We say that p **points to** x and we show it :




# Example Using * and &

- To swap the values of two integer variables we can define the function

```
void swap ( int *a, int *b )
{
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}
```

- Suppose x and y are defined as

```
int  x = 5,  y = 10;
```

To swap them we can call

swap(&x, &y);

# References and Pointers

- A variable can be declared to be a reference to type:

  <type>& <variable>

  i.e.

  ```
  int i;
  int&  p  =  i;                  // p is an alias of the integer object i
  ```

- A reference is  another name for the location it refers to.

  i.e. in the previous example i and p are names of the same location

  ```
  p = 5;
  ```

  causes i to be 5.

- References are usually used for parameter passing .



# Example with References

- The swap function can be written as:

```
void swap( int& a, int& b)
{
  int temp;
  temp = a;
  a = b;
  b = temp;
}
```

Then if we have declare

```
int x = 5, y=10;
```

we can swap them by calling

```
swap(x, y);
```

# Dynamic Data

- C++ has three kinds of data:
  - *automatic data* - allocated, deallocated automatically
  - *static data* - exists throughout the program
  - *dynamic data* - allocated, deallocated by the programmer

- We'll discuss how dynamic data are allocated and deallocated.



# Allocation of Dynamic Data

- When the new operator is applied to a constructor,
  - an object of the class is generated
  - the address of the object is returned

- Example:

```
int* ip, *ap;
ip = new int;
ap = new int[50];

Point * r, *s;
r = new Point(3,4);
s = new Point[20];
```

The last example calls Point()

with each s[i] (i = 0 to 19).

# Deallocation of Dynamic Data

- When the delete operator is applied to a pointer that points to a dynamic object, it frees the space the pointer points to
  - The pointer is then considered unassigned.
- Example:
  - delete ip;
    delete [] ap;
- Good Idea:
  After deleting the space, set the pointer to NULL.
- If the address (pointer value) is NULL and apply delete to it, nothing happens.



# Destructors

- The destructor for a class X is a method with name ~X().
  - It has no arguments and no return type.
  - It cannot be called; it is *invoked* by the system.
- The destructor ~X is invoked:
  - at the end of the scope of a variable of type X
  - when delete is applied to a variable of type X*
  - at the end of a function with an argument of type X
  - to destroy temporaries after their use
  - when a variable with an X member is deleted.
- When the delete operator is applied to a pointer, it
  - invokes the destructor of the class first
  - then recycles the storage pointed to by the pointer.
- If p is an array of n objects of class X, then
  delete [] p ;
  applies the destructor of X to each element of the array p
  then deletes the array p itself.
- A destructor is needed for composite objects that contain dynamic data
  For simple objects like Point, it is not necessary.

# Example: Integer Vector

- An implementation of arrays of integers that
  - check for out-of-range condition
  - can be assigned.
- The code can be found in the examples in following files:

  IntVector Class Header File
  IntVector Class Source File
  IntVector Class Test Driver

**Note:**

- The destructor has to deallocate all the dynamic data allocated by the member functions.



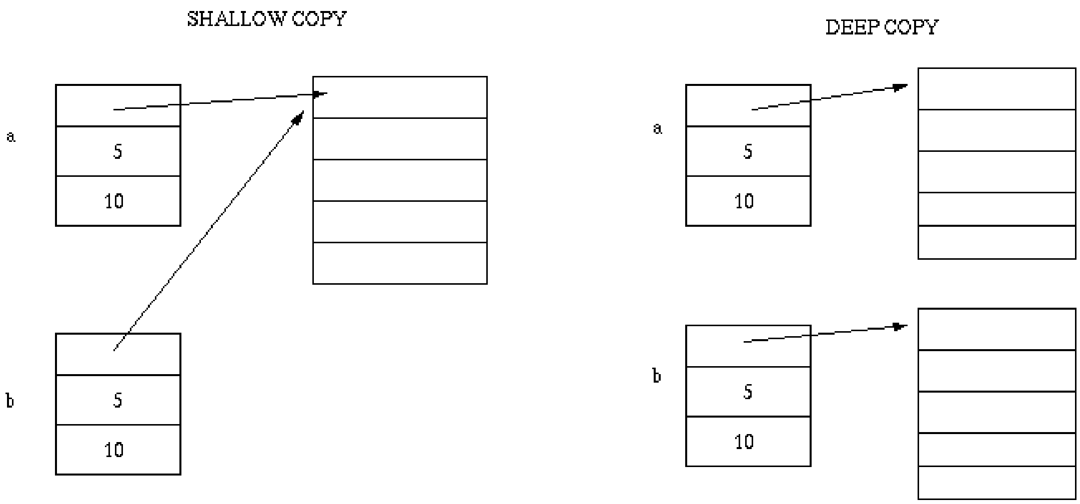
# Copy Constructors (or Copy Initializers)

- A copy initializer for a class C is a constructor of the form

  C(const C&)
- It is used automatically when copying occurs.
- Copying an object occurs
  - at an assignment
  - at initialization
  - passing arguments (by value)
  - returning values from a function
- In the first case, the assignment operator is used. In the rest, the copy constructor is used.
- If no constructor is defined, the system performs a ***shallow copy*** (or ***memberwise copy*** ):
  - only the top members of the object are copied
- If the object has dynamic members
  - a ***deep copy*** is required
  - a copy constructor and an assignment operator must be defined

# Shallow vs. Deep Copy

- Suppose we execute
  IntVector a(5,10);
  IntVector b = a;






# Example: Copy Constructor for IntVector

```
class IntVector
{
 public:

   IntVector();
   // Default constructor

   IntVector( int l, int h );
   // Parameterised constructor

   IntVector( const IntVector&
                     someVector );
   // Copy constructor
 ….
 private:

   int* value;   // Pointer to a dynamically-
                    allocated array of integers
   int  low;     // the lower index bound
   int  high;    // the upper index bound

   void copy( const IntVector& someVector );
   // Helper function used by copy constructor
 }
```

```
IntVector::IntVector( const IntVector& someVector )
   // Copy constructor
{
   copy( someVector );
}


void IntVector::copy( const IntVector& someVector )

   // Private member helper function that copie
   // the contents of IntVector
   // 'someVector' into the current IntVector
{
   low = someVector.low;
   high = someVector.high;
   value = new int[ high - low + 1 ];

   for ( int i = 0; i <= high - low; i++ )
   {
      value[i] = someVector.value[i];
   }
}
```

# Friends

- If we want a function or method f defined outside a class C to have access to the private members of C, we declare (in class C) f to be a ***friend*** of C.

- **Example: Function Friend**
  Declaring distance function to be a friend of Point:

```
class Point {
        friend double distance(Point, Point);
        ....
}
…
double distance(Point a, Point b) {
        return sqrt((a.x - b.x)*(a.x - b.x) + (a.y - b.y)*(a.y - b.y));
}
```

- A class A can also declare another class B to be a friend.
  B can then access the private members of A.

- **Example: Class Friend**
  If we want the class Triangle to have access to Point we define:

```
class Point
{
      friend class Triangle;
.....
```



# Function Overloading

- Use the same name for different (but similar) functions
  - i.e. define the functions length for a string and an array of int

- At a call, C++ chooses one of them according to the argument type as follows:
  - if an exact argument type match is found, the corresponding function is called
  - if a unique match is found after type promotions, the corresponding function is called
  - if a unique match is found after type conversions, the corresponding function is called

# Operator Overloading

- You can redefine in any class the built-in operators:
  `+, - , ..., =, ++, ..., [], <<, ..., etc.`
  Except
  ::, . , sizeof ?:
- An overloaded operator can be defined as:
  - a member function in a class, or
  - a friend of a class
- An overloaded operator:
  - must have at least one class (or struct) argument
  - has the same number of arguments, precedence and associativity as the built-in operation
  - if its first argument is of class X, it can be declared as a member of X; otherwise it should be a non-member.

**Example:**

- In the following we'll use the Complex class (representing complex numbers):
  Complex Class Header
  Complex Class Source
  Complex Class Test Driver
- We'll define a + operator (to add two complex numbers).



# Example: Overloading + in Complex

- As a non member: I
  It should be declared as friend in Complex

```
Complex operator+(Complex x, Complex y)
{
        return Complex(x.realPart + y.realPart,
                x.imaginaryPart +y.imaginaryPart);
}
```

- In this case, we can use it using the functional notation
  c = operator+(a, b);
  or the operator notation
  c = a + b;

- As a member:

```
Complex Complex::operator+( Complex y)
{
        return Complex( realPart + y.realPart,
                imaginaryPart + y.imaginaryPart);
}
```

- In this case, we can use it using the functional notation
  c = a.operator+(b);
  or the operator notation
  c = a + b;

# Example: Overloading [] in IntVector

- Code:

```
int&  IntVector::operator[](int i) {
        if ( i < low || i > high ) {
                cerr << "Index out of range\n";
                exit(1);
        }
        return value[i - low];
}
```

- Then we can do the following:

```
IntVector a(5,10), b(1,20);
a[5] = 0;
b[1] = a[5];
```

- See examples for the complete code (that includes [] and = operators) for IntVector



# Type Conversion Operators

- Built-in types are automatically converted to different Built-in types if this is necessary and possible:
  - during the evaluation of a sub-expression
  - across assignments
  - at a function call (where the expression is the argument)
  - at a function return (where the expression is the return value)
- For built-in types:
  - user can enforce conversions by using type *coercion* or type *casting* of the form:
- type-name( expression )
- E.g.
  char( 5*20 )   converts the result into a character;
  (int *)( p+2)   converts the result into a pointer to an integer.
- For defined classes,
  - an explicit conversion from class X to Y can be performed by defining in Y a constructor of the form Y(X) or Y(X&).

# Example

Suppose in the class Complex we define:

```
class Complex {
    public:
        Complex()
        // regular constructor
        {
            realPart = 0; imaginaryPart = 0;
        }
        Complex( double a, double b )
        // regular constructor
        {
            realPart = a; imaginaryPart = b
        }
        Complex( double d )
        // typecasts double to Complex
        {
            realPart = d; imaginaryPart = 0
        }
        . . .
};
```

Then we can do:

```
Complex C;
double d = 5;
C = Complex(d);
```



# Variable Kinds

- A C++ program may have many source files with many classes and functions.
- A variable has two characteristics:
  - *extent* or *lifetime*
  - *visibility* or *scope*
- C++ has 4 types of scope:
  - *local (internal)*
    - variable is known only inside the block in which it is declared.
  - *global (external)*
    - variable is declared outside of any function or block
    - is known throughout any source file in which it is declared. i.e. extern variables, functions and classes
  - *file*
    - its scope is the source file in which it is declared
    - it cannot be used in any other file, i.e. static variables
  - *class* or *struct*
    - the scope of a variable which is a member of a class or structure is that class or structure
- A variable can have one of the following 3 extent types:
  - *automatic*
    - created whenever the block in which it is declared is executed;
    - destroyed whenever the block is exited.
  - *dynamic*
    - its memory is explicitly allocated and destroyed using new and delete
  - *static*
    - created when program is loaded in memory and retains its value until the end of the program (i.e. retains its value between calls)
- The scope of a static name **cannot** exceed the source file in which it is declared.

# Example

- Consider the following program ;

```
int k;
static int i;
void p( int a )
{    static int b = 50;
     int c;
     int *w = new int;
     c = a + a;
     b += c;
}

static void q( void )
{    k = 1;
     p( k );
}

void main()
{    int k = 2;
     p( k );
}
```

The characteristics of the variables are summarized in the following table

| | local | file | global |
|---|---|---|---|
| automatic | a,c,w,k (in main) | ---------------- | k (first line),p |
| static | b | i, q | --------------------- |
| dynamic | | | block ref'd by w |



# Memory Organization

- There are a number of 'segments' in memory when a C++ program is running. The following segments are standard:
- *Code Segment :*
  - also known as 'text' segment
  - it contains machine language code produced by the compiler
  - contains constant data
    - e.g. const double e = 2.71828182845905
- *Static Data Segment:*
  - contains all data that have been declared static
  - they are allocated when the program is loaded, and remain in memory until the program terminates
  - Only static *data* are allocated in the data segment (i.e. static functions are not)
  - For instance, if we define

```
static int k;
static void foo(void)
{
     static int i = 1;
     ....
}
```

i and k will be allocated in this segment.

# Memory Organization (con't)

- *Stack Segment*
  - it contains automatic variables (including arguments) and
  - bookkeeping information as below
- *Heap Segment*
  - consists of (dynamic) storage allocated via new
  - data stored in heap segment are not named; access is indirect via pointers
- Pictorially,

| |
|---|
| Code (Read Only) |
| Static (Read Write) |
| Heap (RW) |
| free (RW) |
| Stack (RW) |



# The Run-time Stack

- When a procedure is called the system builds a *stack frame* for the call and pushes it on the top of the stack
- When the procedure returns, the system pops the frame from the stack
- A stack frame contains space for:
  - parameters
  - local automatic variables
  - return value
  - return address
  - other bookkeeping information
  - a pointer to the previous frame

# Example

- Suppose foo is defined as:

```
int foo(int a)
{       static int b = 0;
        int c = a;
        b += a;
        return a + b + c;
 }
```

- When **foo(5)** is called, the stack frame might look like that on the right
- Recursion is done by pushing an additional frame onto the stack. Some C compilers 'recognize' tail recursion and do not push an additional frame.

| |
|---|
| return address |
| 5 (a) |
| 5 (c) |
| return value |
| ... |
| pointer to previous frame |



# Static Var's and the Stack

- Static variables are NOT stored on the stack:
  - they are stored in the static segment
  - they are allocated and initialized exactly once
- **Example**: Consider the following function.

```
int count()
{  static int j = 0;
   j++;
   return j;
}
int main()
{  cout << "first " << count()<< " second " << count()<< " third " <<
        count()<< "/n";
}
```

This produces the output:

first 1 second 2 third 3

which clearly shows that only one location is used for j.

# Dynamic Memory Allocation

- Often it is necessary for the program to allocate memory as it is needed:
  - data size is not known in advance
  - size of data varies widely
  - most objects need to be created at run time
  - most objects need to be dynamic
- This is called *dynamic memory allocation* and is done by operators new and delete (as we have seen).
- Dynamic data created this way
  - is on the heap
  - is accessible through pointers only
  - has dynamic extent
  - is accessible by any part of the program (as long as the address of the data is known)



# New Operator

- its argument is a type (or class)
- it searches the heap to find a space big enough for an object of the given type
- if there is enough space
  - it claims the space found and marks it as occupied
  - returns the location's address
- if there is not enough space
  - an exception is raised or a NULL is returned (depends onthe implementation);
- new can be used with two arguments: a type and a positive integer n, in which case it allocates space for n objects of this type
- i.e.

```
p = new Point(3,4);      // one point
a = new Point[100];      // 100 points
```

# Delete Operator

- its argument is a pointer
- it frees the space occupied by the object the pointer points to
- its argument remains unchanged
- if its argument is NULL, it does nothing
- it will cause an error if you delete a space not previously allocated by the new operator
- if par is a dynamic array of pointers to some type

  delete [] par;

  will apply delete to each element of par before it deletes the space par points to.



# Memory Fragmentation

- Inherent in dynamic allocation is the concept of *fragmentation*.
- Heap is divided into a large number of small fragments.
- Degree of fragmentation depends on the allocation strategy in use. Three allocation strategies are:
  - *first fit* : find the first suitable segment
  - *best fit* : find the smallest suitable segment
  - *worst fit* : find the largest suitable segment

# Dangling Pointers

- A *dangling pointer* is a pointer whose value is the address of a storage block which has been freed.
- Consider the code on the right
  - The error here is in the last line
  - We can't dereference p, since freeing the node it pointed to made p invalid (dangling pointer)
  - The effect is indeterminate.
- The second box on the right shows a case in which more than 1 pointer points to the same object
  - it might not be at all apparent that q is invalid

```
int * p;
p = new int;
. . .
*p = 20;
. . .
delete p;
. . .
*p = 5;
```

```
int * p;
p = new int;
. . .
*p = 20;
int* q = p;
. . .
delete p;
. . .
*q = 5;
```



# Inaccessible Objects - Memory Leaks

- A dynamic object with no pointer pointing to it is *inaccessible*.
- A *memory leak* is created by:
  - inaccessible objects
  - dynamic objects that are not needed anymore (and never deleted)
- In certain languages, objects cannot be explicitly freed. These languages have a garbage collection or scavenging system that frees objects no longer in use.
- In C and C++, garbage collecting is done by the programmer, whereas in Java and Scheme, the system performs the garbage collection.