AN OVERVIEW OF C++

OBJECTIVES

- Introduction
- What is object-oriented programming?
- Two versions of C++
- C++ console I/O
- C++ comments
- Classes: A first look
- Some differences between C and C++
- Introducing function overloading
- C++ keywords
- Introducing Classes

INTRODUCTION

- C++ is the C programmer's answer to Object-Oriented Programming (OOP).
- C++ is an enhanced version of the C language.
- C++ adds support for OOP without sacrificing any of C's power, elegance, or flexibility.
- C++ was invented in 1979 by Bjarne Stroustrup at Bell Laboratories in Murray Hill, New Jersey, USA.

INTRODUCTION (CONT.)

- The elements of a computer language do not exist in a void, separate from one another.
- The features of C++ are highly integrated.
- Both object-oriented and non-object-oriented programs can be developed using C++.

WHAT IS OOP?

- OOP is a powerful way to approach the task of programming.
- OOP encourages developers to decompose a problem into its constituent parts.
- Each component becomes a self-contained object that contains its own instructions and data that relate to that object.
- So, complexity is reduced and the programmer can manage larger programs.

WHAT IS OOP? (CONT.)

- All OOP languages, including C++, share three common defining traits:
  - Encapsulation
    - Binds together code and data
  - Polymorphism
    - Allows one interface, multiple methods
  - Inheritance
    - Provides hierarchical classification
    - Permits reuse of common code and data

TWO VERSIONS OF C++

- A traditional-style C++ program -

```
#include <iostream.h>

int main()
{
        /* program code */
        return 0;
}
```

TWO VERSIONS OF C++ (CONT.)

- A modern-style C++ program that uses the new-style headers and a namespace -

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

int main()
{
        /* program code */
        return 0;
}
```

THE NEW C++ HEADERS

- The new-style headers do not specify filenames.
- They simply specify standard identifiers that might be mapped to files by the compiler, but they need not be.
  - <iostream>
  - <vector>
  - <string>, not related with <string.h>
  - <cmath>, C++ version of <math.h>
  - <cstring>, C++ version of <string.h>
- Programmer defined header files should end in ".h".

SCOPE RESOLUTION OPERATOR (::)

- Unary Scope Resolution Operator
  - Used to access a hidden global variable
  - **Example:** usro.cpp
- Binary Scope Resolution Operator
  - Used to associate a member function with its class (will be discussed shortly)
  - Used to access a hidden class member variable (will be discussed shortly)
  - **Example:** bsro.cpp

NAMESPACES

- A namespace is a declarative region.
- It localizes the names of identifiers to avoid name collisions.
- The contents of new-style headers are placed in the **std** namespace.
- A newly created class, function or global variable can put in an existing namespace, a new namespace, or it may not be associated with any namespace
  - In the last case the element will be placed in the global unnamed namespace.
- Example: namespace.cpp

C++ CONSOLE I/O (OUTPUT)

- cout << “Hello World!”;
  - printf(“Hello World!”);
- cout << iCount; /* int iCount *
  - printf(“%d”, iCount);
- cout << 100.99;
  - printf(“%f”, 100.99);
- cout << “\n”, or cout << ‘\n’, or endl
  - printf(“\n”)
- In general, cout << *expression*;

cout ???

Shift right operator ???

How does a shift right operator produce output to the screen?

Do we smell polymorphism here???

C++ CONSOLE I/O (INPUT)

- cin >> strName; /* char strName[16] */
  - scanf("%s", strName);
- cin >> iCount; /* int iCount */
  - scanf("%d", &iCount);
- cin >> fValue; /* float fValue */
  - scanf("%f", &fValue);
- In general, cin >> *variable*;

Hmmm. Again polymorphism.

C++ CONSOLE I/O (I/O CHAINING)

- cout << “Hello” << ‘ ‘ << “World” << ‘!’;
- cout << “Value of iCount is: ” << iCount;
- cout << “Enter day, month, year: ”;
  - cin >> day >> month >> year;
    - cin >> day;
    - cin >> month;
    - cin >> year

What’s actually happening here? Need to learn more.

C++ CONSOLE I/O (EXAMPLE)

```
include <iostream>
int main()
{
    char str[16];
    std::cout << “Enter a string: ”;
    std::cin >> str;
    std::cout << “You entered: ”
                        << str;
}
```

```
include <iostream>
using namespace std;
int main()
{
    char str[16];
    cout << “Enter a string: ”;
    cin >> str;
    cout << “You entered: ”
                        << str;
}
```

C++ COMMENTS

- Multi-line comments
  - /* one or more lines of comments */
- Single line comments
  - // …

CLASSES: A FIRST LOOK

- General syntax -

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

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

CLASSES: A FIRST LOOK (CONT.)

- A class declaration is a logical abstraction that defines a new type.
- It determines what an object of that type will look like.
- An object declaration creates a physical entity of that type.
- That is, an object occupies memory space, but a type definition does not.
- **Example:** p-23.cpp, p-26.cpp, stack-test.c.

CLASSES: A FIRST LOOK (CONT.)

- Each object of a class has its own copy of every variable declared within the class (except static variables which will be introduced later), but they all share the same copy of member functions.
  - How do member functions know on which object they have to work on?
    - The answer will be clear when "***this***" pointer is introduced.

SOME DIFFERENCES BETWEEN C AND C++

- No need to use "void" to denote empty parameter list.
- All functions must be prototyped.
- If a function is declared as returning a value, it ***must*** return a value.
- Return type of all functions must be declared explicitly.
- Local variables can be declared anywhere.
- C++ defines the **bool** datatype, and keywords **true** (any nonzero value) and **false** (zero).

INTRODUCING FUNCTION OVERLOADING

- Provides the mechanism by which C++ achieves one type of polymorphism (called **compile-time polymorphism**).
- Two or more functions can share the same name as long as either
  - The type of their arguments differs, or
  - The number of their arguments differs, or
  - Both of the above

INTRODUCING FUNCTION OVERLOADING (CONT.)

- The compiler will automatically select the correct version.
- The return type alone is not a sufficient difference to allow function overloading.
- **Example:** p-34.cpp, p-36.cpp, p-37.cpp.

Q. Can we confuse the compiler with function overloading?
A. Sure. In several ways. Keep exploring C++.

C++ KEYWORDS (PARTIAL LIST)

- bool
- catch
- delete
- false
- friend
- inline
- namespace
- new
- operator
- private
- protected
- public
- template
- this
- throw
- true
- try
- using
- virtual
- wchar_t

INTRODUCING CLASSES

CONSTRUCTORS

- Every object we create will require some sort of initialization.
- A class's constructor is automatically called by the compiler each time an object of that class is created.
- A constructor function has the ***same name*** as the class and has ***no return type***.
- There is no explicit way to call the constructor.

DESTRUCTORS

- The complement of a constructor is the destructor.
- This function is automatically called by the compiler when an object is destroyed.
- The name of a destructor is the ***name of its class***, preceded by a ~.
- There is explicit way to call the destructor but highly discouraged.
- **Example** : cons-des-0.cpp

CONSTRUCTORS & DESTRUCTORS

- 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.
- **Example**: cons-des-1.cpp

CONSTRUCTORS & DESTRUCTORS

- Constructors and destructors are typically declared as **public**.
- That is why the compiler can call them when an object of a class is declared anywhere in the program.
- If the constructor or destructor function is declared as **private** then no object of that class can be created outside of that class**. *What type of error ?***
- **Example**: private-cons.cpp, private-des.cpp

CONSTRUCTORS THAT TAKE PARAMETERS

- It is possible to ***pass arguments*** to a constructor function.
- Destructor functions ***cannot*** have parameters.
- A constructor function with no parameter is called the ***default constructor*** and is supplied by the compiler automatically if no constructor defined by the programmer.
- The compiler supplied default constructor ***does not initialize*** the member variables to any default value; so they contain garbage value after creation.
- Constructors ***can be overloaded***, but destructors ***cannot be overloaded.***
- A class can have multiple constructors.
- **Example**: cons-des-3.cpp, cons-des-4.cpp, cons-des-5.cpp, cons-des-6.cpp

OBJECT POINTERS

- It is possible to access a member of an object via a pointer to that object.
- Object pointers play a massive role in run-time polymorphism (will be introduced later).
- When a pointer is used, the arrow operator (->) rather than the dot operator is employed.
- Just like pointers to other types, an object pointer, when incremented, will point to the next object of its type.
- **Example**: obj.cpp

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. ***Why ?***
- 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;
}
```

- 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.
- Restrictions
  - Same as normal in-line functions.

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;
}
```

LECTURE CONTENTS

- Teach Yourself C++
  - Chapter 1 (Full, with exercises)
  - Chapter 2.1, 2,2, 2.4, 2.6, 2.7

A CLOSER LOOK AT CLASSES

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.
- Solution: **Copy constructor** (to be discussed later)
- **Example:** assign-object.cpp

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.
- **Examples:** From book

PASSING OBJECTS TO FUNCTIONS

- In call by reference, as no new objects are formed, constructors and destructors are not called.
- But in call value, while making a copy, <u>constructors are not called</u> for the copy but <u>destructors are called</u>.
- Can this cause any problem in any case?
- Yes. Solution: **Copy constructor** (discussed later)
- **Example**: obj-passing1.cpp, obj-passing2.cpp, obj-passing-problem.cpp

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, <u>constructors are not called</u> for the copy but <u>destructors are called</u>
- After the value has been returned, this object is destroyed.
- The destruction of this temporary object might cause unexpected side effects in some situations.
- Solution: **Copy constructor** (to be discussed later)
- **Example:** ret-obj-1.cpp, ret-obj-2.cpp, ret-obj-3.cpp

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 <u>friend</u>.
- 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;
   // can access private
   member 'a' directly
  MyClass obj2(100);
  cout << obj2.a << endl;
}
void main()
 {
  MyClass o1(10);
  ff1(o1);
}
```

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 <u>an object</u> of that class. The object can be <u>declared within or passed</u> to the friend function.
- <u>A member function can directly access class members</u>.
- A function can be a member of one class and a friend of another.
- **Example** : friend1.cpp, friend2.cpp, friend3.cpp

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

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

FRIEND FUNCTIONS

```
class YourClass; // a forward
    declaration
class MyClass {
   int a; // private member
public:
   MyClass(int a1) { a = a1; }
   int compare (YourClass obj) {
      return (a – obj.a)
   }
};
```

```
class YourClass {
   int a; // private member
public:
   YourClass(int a1) { a = a1; }
   friend int MyClass::compare
    (YourClass obj);
};
void main() {
   MyClass o1(10); Yourclass
    o2(5);
   int n = o1.compare(o2); // n = 5
}
```

CONVERSION FUNCTION

- Used to convert an object of one type into an object of another type.
- A conversion function automatically converts an object into a value that is compatible with the type of the expression in which the object is used.
- General form: *operator type() {return value;}*
- *type* is the target type and *value* is the value of the object after conversion.
- No parameter can be specified.
- Must be a member of the class for which it performs the conversion.
- **Examples**: From Book.

CONVERSION FUNCTION

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

class coord
{
    int x, y;
public:
    coord(int i, int j){ x = i; y = j; }
    operator int() { return x*y; }
};
```

```
int main
{
    coord o1(2, 3), o2(4, 3);
    int i;

    i = o1;
    // automatically converts to integer
    cout << i << '\n';

    i = 100 + o2;
    // automatically converts to integer
    cout << i << '\n';

    return 0;
}
```

CONVERSION FUNCTION

- Suppose we have the following two classes:
  - Cartesian Coordinate: CCoord
  - Polar Coordinate: PCoord

- Can we use conversion function to perform conversion between them?

```
CCoord c(10, 20);
PCoord p(15, 120);

p = c;
c = p;
```

STATIC CLASS MEMBERS

- A class member can be declared as ***static***
- Only one copy of a ***static*** variable exists – no matter how many objects of the class are created
  - All objects share the same variable
- It can be private, protected or public
- A ***static*** member variable exists before any object of its class is created
- In essence, a ***static*** class member is a global variable that simply has its scope restricted to the class in which it is declared

STATIC CLASS MEMBERS

- When we declare a ***static*** data member within a class, we are not defining it
- Instead, we must provide a definition for it elsewhere, outside the class
- To do this, we re-declare the ***static*** variable, using the scope resolution operator to identify which class it belongs to
- All ***static*** member variables are initialized to **0** by default

STATIC CLASS MEMBERS

- The principal reason ***static*** member variables are supported by C++ is to avoid the need for global variables
- Member functions can also be ***static***
  - Can access only other ***static*** members of its class directly
  - Need to access ***non-static*** members through an object of the class
  - Does not have a ***this*** pointer
  - Cannot be declared as ***virtual***, ***const*** or ***volatile***
- ***static*** member functions can be accessed through an object of the class or can be accessed independent of any object, via the class name and the scope resolution operator
  - Usual access rules apply for all ***static*** members
- **Example**: static.cpp

STATIC CLASS MEMBERS

```
class myclass {
   static int x;
public:
   static int y;
   int getX() { return x; }
   void setX(int x) {
      myclass::x = x;
   }
};
int myclass::x = 1;
int myclass::y = 2;
```

```
void main ( ) {
   myclass ob1, ob2;
   cout << ob1.getX() << endl; // 1
   ob2.setX(5);
   cout << ob1.getX() << endl; // 5
   cout << ob1.y << endl; // 2
   myclass::y = 10;
   cout << ob2.y << endl; // 10
   // myclass::x = 100;
   // will produce compiler error
}
```

CONST MEMBER FUNCTIONS AND *MUTABLE*

- When a class member is declared as *const* it can't modify the object that invokes it.
- A *const* object can't invoke a non-*const* member function.
- But a *const* member function can be called by either *const* or non-*const* objects.
- If you want a *const* member function to modify one or more member of a class but you don't want the function to be able to modify any of its other members, you can do this using *mutable*.
- *mutable* members can modified by a *const* member function.
- **Examples**: From Book.

LECTURE CONTENTS

- **Teach Yourself C++**
  - Chapter 3 (Full, with exercises)
  - Chapter 13 (13.2,13.3 and 13.4)

ARRAYS, POINTERS AND REFERENCES

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**.
- If **A** does not have a default constructor, then the above array declaration statement will produce compiler error.

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) };**

USING POINTERS TO OBJECTS

- 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.

THIS POINTER

- A special pointer in C++ that points to the object that generates the call to the method
- Let,
  - ***class A{ public: void f1() { … } };***
  - **A ob; ob.f1();**
- 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.
- It also automatically calls the member function with the address of the object through which the function is invoked.
- So the above example works as follows –
  - ***class A{ public: void f1(A *this) { … } };***
  - **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
  }
```

```
void f1() {
      int x = 0;
      cout << x; // prints value of local 'x'
      cout << this->x; // prints value of member 'x'
  }
};
```

USING NEW AND DELETE

- C++ introduces two operators for dynamically allocating and deallocating memory :
  - ***p_var = new type***
  - new returns a pointer to dynamically allocated memory that is sufficient to hold a data obect of type *type*
  - ***delete p_var***
  - releases the memory previously allocated by new
- Memory allocated by new must be released using delete
- The lifetime of an object is directly under our control and is unrelated to the block structure of the program

USING <u>NEW</u> AND <u>DELETE</u>

- 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
  - No need to use ***sizeof*** operator while using new.
  - New automatically returns a pointer of the specified type.
  - In case of objects, new calls dynamically allocates the object and call its constructor
  - In case of objects, delete calls the destructor of the object being released

USING <u>NEW</u> AND <u>DELETE</u>

- 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.*

USING 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.

USING <u>NEW</u> AND <u>DELETE</u>

- 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 [ ]*

USING 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;
```

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

- **Using reference -**

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

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

- **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

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);
}
```

```
void main() {
  myclass obj;
  cout << obj.getx() << endl;
  f(obj);
  cout << obj.getx() << endl;
}
```

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

Output:

1

100

2

So, here f() can be used to both set the value of x and read the value of x

Example: From Book(151 – 153)

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**

LECTURE CONTENTS

- Teach Yourself C++
  - Chapter 4 (See All Exercise)

FUNCTION OVERLOADING

Chapter 5

Department of CSE, BUET

OBJECTIVES

- Overloading Constructor Functions
- Creating and Using a Copy Constructor
- The **overload** Anachronism (not in syllabus)
- Using Default Arguments
- Overloading and Ambiguity
- Finding the address of an overloaded function

OVERLOADING CONSTRUCTOR FUNCTIONS

- It is possible to overload constructors, but destructors cannot be overloaded.
- Three main reasons to overload a constructor function
  - To gain flexibility
  - To support arrays
  - To create copy constructors
- There must be a constructor function for each way that an object of a class will be created, otherwise compile-time error occurs.

OVERLOADING CONSTRUCTOR FUNCTIONS (CONTD.)

- Let, we want to write
  - MyClass ob1, ob2(10);
- Then MyClass should have the following two constructors (it may have more)
  - MyClass ( ) { … }
  - MyClass (int n) { … }
- Whenever we write a constructor in a class, the compiler does not supply the default no argument constructor automatically.
- No argument constructor is also necessary for declaring arrays of objects without any initialization.
  - MyClass array1[5]; // uses MyClass () { … } for each element
- But with the help of an overloaded constructor, we can also initialize the elements of an array while declaring it.
  - MyClass array2[3] = {1, 2, 3} // uses MyClass (int n) { … } for each element

OVERLOADING CONSTRUCTOR FUNCTIONS (CONTD.)

- Overloading constructor functions also allows the programmer to select the most convenient method to create objects.
  - Date d1(22, 9, 2007); // uses Date(int d, int m, int y)
  - Date d2("22-Sep-2007"); // uses Date(char* str)
- Another reason to overload a constructor function is to support dynamic arrays of objects created using "new".
- As dynamic arrays of objects cannot be initialized, the class must have a no argument constructor to avoid compiler error while creating dynamic arrays using "new".

CREATING AND USING A COPY CONSTRUCTOR

- By default when a assign an object to another object or initialize a new object by an existing object, a bitwise copy is performed.
- This cause problems when the objects contain pointer to dynamically allocated memory and destructors are used to free that memory.
- It causes the same memory to be released multiple times that causes the program to crash.
- Copy constructors are used to solve this problem while we perform object initialization with another object of the same class.
  - MyClass ob1;
  - MyClass ob2 = ob1; // uses copy constructor
- Copy constructors do not affect assignment operations.
  - MyClass ob1, ob2;
  - ob2 = ob1; // does not use copy constructor

CREATING AND USING A COPY CONSTRUCTOR (CONTD.)

- If we do not write our own copy constructor, then the compiler supplies a copy constructor that simply performs bitwise copy.
- We can write our own copy constructor to dictate precisely how members of two objects should be copied.
- The most common form of copy constructor is
  - classname (const classname &obj) {
  -     // body of constructor
  - }

CREATING AND USING A COPY CONSTRUCTOR (CONTD.)

- Object initialization can occur in three ways
  - When an object is used to initialize another in a declaration statement
    - MyClass y;
    - MyClass x = y;
  - When an object is passed as a parameter to a function
    - func1(y); // calls "void func1(MyClass obj)"
  - When a temporary object is created for use as a return value by a function
    - y = func2(); // gets the object returned from "MyClass func2()"
- See the examples from the book and the supplied codes to have a better understanding of the activities and usefulness of copy constructors.
  - Example: copy-cons.cpp

USING DEFAULT ARGUMENTS

- It is related to function overloading.
  - Essentially a shorthand form of function overloading
- It allows to give a parameter a default value when no corresponding argument is specified when the function is called.
  - void f1(int a = 0, int b = 0) { … }
  - It can now be called in three different ways.
    - f1(); // inside f1() 'a' is '0' and b is '0'
    - f1(10); // inside f1() 'a' is '10' and b is '0'
    - f1(10, 99); // inside f1() 'a' is '10' and b is '99'
  - We can see that we cannot give 'b' a new (non-default) value without specifying a new value for 'a'.
  - So while specifying non-default values, we have to start from the leftmost parameter and move to the right one by one.

USING DEFAULT ARGUMENTS (CONTD.)

- Default arguments must be specified only once: either in the function's prototype or in its definition.
- All default parameters must be to the right of any parameters that don't have defaults.
  - void f2(int a, int b = 0); // no problem
  - void f3(int a, int b = 0, int c = 5); // no problem
  - void f4(int a = 1, int b); // compiler error
- So, once you begin to define default parameters, you cannot specify any parameters that have no defaults.
- Default arguments must be constants or global variables. They cannot be local variables or other parameters.

USING DEFAULT ARGUMENTS (CONTD.)

- Relation between default arguments and function overloading.
  - void f1(int a = 0, int b = 0) { … }
  - It acts as the same way as the following overloaded functions –
    - void f2( ) { int a = 0, b = 0; … }
    - void f2(int a) { int b = 0; … }
    - void f2(int a, int b) { … }
- Constructor functions can also have default arguments.

USING DEFAULT ARGUMENTS (CONTD.)

- It is possible to create copy constructors that take additional arguments, as long as the additional arguments have default values.
  - MyClass(const MyClass &obj, int x = 0) { … }
- This flexibility allows us to create copy constructors that have other uses.
- See the examples from the book to learn more about the uses of default arguments.

OVERLOADING AND AMBIGUITY

- Due to automatic type conversion rules.
- Example 1:
  - void f1(float f) { … }
  - void f1(double d) { … }
  - float x = 10.09;
  - double y = 10.09;
  - f1(x); // unambiguous – use f1(float)
  - f1(y); // unambiguous – use f1(double)
  - f1(10); // ambiguous, compiler error
    - Because integer '10' can be promoted to both "float" and "double".

OVERLOADING AND AMBIGUITY (CONTD.)

- Due to the use of reference parameters.
- Example 2:
  - void f2(int a, int b) { … }
  - void f2(int a, int &b) { … }
  - int x = 1, y = 2;
  - f2(x, y); // ambiguous, compiler error

Department of CSE, BUET

OVERLOADING AND AMBIGUITY (CONTD.)

- Due to the use of default arguments.
- Example 3:
  - void f3(int a) { … }
  - void f3(int a, int b = 0) { … }
  - f3(10, 20); // unambiguous – calls f3(int, int)
  - f3(10); // ambiguous, compiler error

FINDING THE ADDRESS OF AN OVERLOADED FUNCTION

- Example:
  - void space(int a) { … }
  - void space(int a, char c) { … }
  - void (*fp1)(int);
  - void (*fp2)(int, char);
  - fp1 = space; // gets address of space(int)
  - fp2 = space; // gets address of space(int, char)
- So, it is the declaration of the pointer that determines which function's address is assigned.

LECTURE CONTENTS

- Teach Yourself C++
  - Chapter 5 (Full, with exercises)
    - Except "The **overload** Anachronism"

INHERITANCE

Chapter 7

OBJECTIVES

- Base class access control
- Using **protected** members
- Visibility of base class members in derived class
- Constructors, destructors, and inheritance
- Multiple inheritance
- Virtual base classes

BASE CLASS ACCESS CONTROL

- class derived-class-name : ***access*** base-class-name { … };
- Here ***access*** is one of three keywords
  - public
  - private
  - protected
- Use of ***access*** is optional
  - It is private by default if the derived class is a **class**
  - It is public by default if the derived class is a **struct**

USING PROTECTED MEMBERS

- Cannot be directly accessed by non-related classes and functions
- But can be directly accessed by the derived classes
- Can also be used with structures

VISIBILITY OF BASE CLASS MEMBERS IN DERIVED CLASS

- When a class (derived) inherits from another (base) class, the visibility of the members of the base class in the derived class is as follows.

	Member visibility in derived class		
	Type of Inheritance		
Member access specifier in base class	Private	Protected	Public
Private	Not Inherited	Not Inherited	Not Inherited
Protected	Private	Protected	Protected
Public	Private	Protected	Public

CONSTRUCTORS, DESTRUCTORS, AND INHERITANCE

- Both base class and derived class can have constructors and destructors.
- Constructor functions are executed in the order of derivation.
- Destructor functions are executed in the reverse order of derivation.
- While working with an object of a derived class, the base class constructor and destructor are always executed no matter how the inheritance was done (private, protected or public).

CONSTRUCTORS, DESTRUCTORS, AND INHERITANCE (CONTD.)

```
class base {
public:
   base() {
      cout << "Constructing base class\n";
   }
   ~base() {
      cout << "Destructing base class\n";
   }
};
class derived : public base {
public:
   derived() {
      cout << "Constructing derived class\n";
   }
   ~derived() {
      cout << "Destructing derived class\n";
   }
};
```

```
void main() {
   derived obj;
}
```

Output:

- Constructing base class
- Constructing derived class
- Destructing derived class
- Destructing base class

CONSTRUCTORS, DESTRUCTORS, AND INHERITANCE (CONTD.)

- If a base class constructor takes parameters then it is the responsibility of the derived class constructor(s) to collect them and pass them to the base class constructor using the following syntax -
  - derived-constructor(arg-list) : base(arg-list) { … }
  - Here "base" is the name of the base class
- It is permissible for both the derived class and the base class to use the same argument.
- It is also possible for the derived class to ignore all arguments and just pass them along to the base class.

CONSTRUCTORS, DESTRUCTORS, AND INHERITANCE (CONTD.)

```
class MyBase {
public:
  int x;
  MyBase(int m) { x = m; }
};
class MyDerived : public MyBase {
public:
  int y;
  MyDerived() : MyBase(0) { y = 0; }
  MyDerived(int a) : MyBase(a)
  {
    y = 0;
  }
  MyDerived(int a, int b) : MyBase(a)
  {
    y = b;
  }
};
```

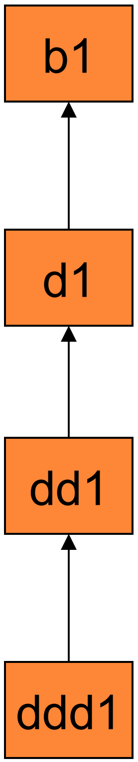
```
void main() {
  MyDerived o1; // x = 0, y = 0
  MyDerived o2(5); // x = 5, y = 0
  MyDerived o3(6, 7); // x = 6, y = 7
}
```

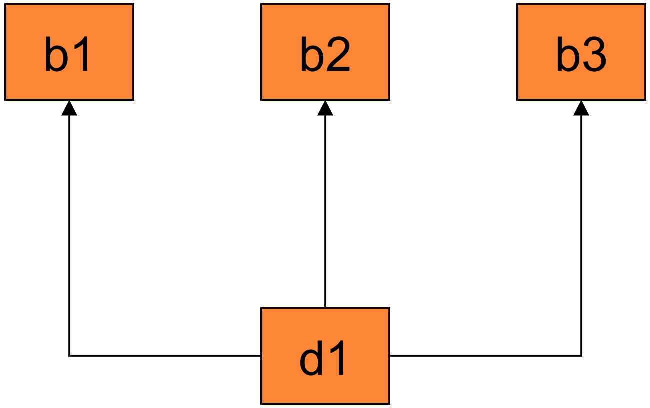
- As "MyBase" does not have a default (no argument) constructor, every constructor of "MyDerived" must pass the parameters required by the "MyBase" constructor.

MULTIPLE INHERITANCE

- A derived class can inherit more than one base class in two ways.
  - Option-1: By a chain of inheritance
    - b1 -> d1 -> dd1 -> ddd1 -> …
    - Here b1 is an indirect base class of both dd1 and ddd1
    - Constructors are executed in the order of inheritance
    - Destructors are executed in the reverse order
  - Option-2: By directly inheriting more than one base class
    - class d1 : *access* b1, *access* b2, …, *access* bN { … }
    - Constructors are executed in the order, left to right, that the base classes are specified
    - Destructors are executed in the reverse order

MULTIPLE INHERITANCE (CONTD.)





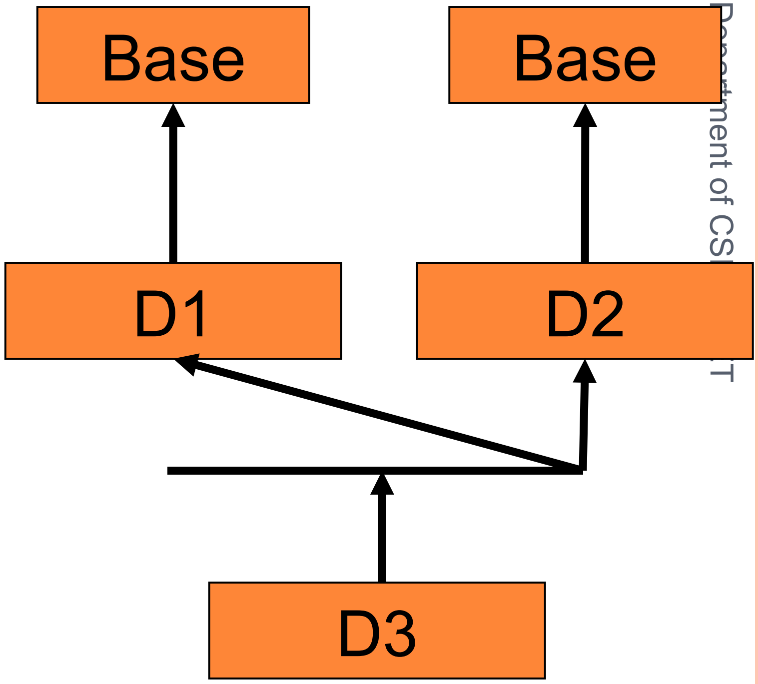
Option - 1

Option - 2

Department of CSE, BUET

VIRTUAL BASE CLASSES

- Consider the situation shown.
- Two copies of *Base* are included in *D3*.
- This causes ambiguity when a member of *Base* is directly used by *D3*.



Department of CSE, IT

VIRTUAL BASE CLASSES (CONTD.)

- class Base {
- public:
-    int i;
- };
- class D1 : public Base {
- public:
-    int j;
- };
- class D2 : public Base {
- public:
-    int k;
- };

- class D3 : public D1, public D2 {
-   // contains two copies of ‘i’
- };
- void main() {
-    D3 obj;
-    obj.i = 10; // ambiguous, compiler error
-    obj.j = 20; // no problem
-    obj.k = 30; // no problem
-    obj.D1::i = 100; // no problem
-    obj.D2::i = 200; // no problem
- }

VIRTUAL BASE CLASSES (CONTD.)

- class Base {
- public:
-    int i;
- };
- class D1 : **virtual** public Base {
- public:
-    int j;
- }; // activity of D1 not affected
- class D2 : **virtual** public Base {
- public:
-    int k;
- }; // activity of D2 not affected

- class D3 : public D1, public D2 {
-   // contains only one copy of 'i'
- }; // no change in this class definition
- void main() {
-    D3 obj;
-    obj.i = 10; // no problem
-    obj.j = 20; // no problem
-    obj.k = 30; // no problem
-    obj.D1::i = 100; // no problem, overwrites '10'
-    obj.D2::i = 200; // no problem, overwrites '100'
- }

LECTURE CONTENTS

- Teach Yourself C++
  - Chapter 7 (Full, with exercise)
  - Study the examples from the book carefully

VIRTUAL FUNCTIONS

Chapter 10

OBJECTIVES

- Polymorphism in C++
- Pointers to derived classes
- Important point on inheritance
- Introduction to virtual functions
- Virtual destructors
- More about virtual functions
- Final comments
- Applying polymorphism

POLYMORPHISM IN C++

- 2 types
  - Compile time polymorphism
    - Uses static or early binding
    - Example: Function and operator overloading
  - Run time polymorphism
    - Uses dynamic or early binding
    - Example: Virtual functions

POINTERS TO DERIVED CLASSES

- C++ allows base class pointers to point to derived class objects.
- Let we have –
  - class base { … };
  - class derived : public base { … };
- Then we can write –
  - base *p1; derived d_obj; p1 = &d_obj;
  - base *p2 = new derived;

POINTERS TO DERIVED CLASSES (CONTD.)

- Using a base class pointer (pointing to a derived class object) we can access only those members of the derived object **that were inherited from the base**.
  - It is different from the behavior that Java shows.
  - We can get Java-like behavior using virtual functions.
- This is because the **base pointer** has knowledge only of the base class.
- It knows nothing about the members added by the derived class.

POINTERS TO DERIVED CLASSES (CONTD.)

- class base {
- public:
-   void show() {
-     cout << “base\n”;
-   }
- };
- class derived : public base {
- public:
-   void show() {
-     cout << “derived\n”;
-   }
- };

- void main() {
-   base b1;
-   b1.show(); // base
-   derived d1;
-   d1.show(); // derived
-   base *pb = &b1;
-   pb->show(); // base
-   **pb = &d1;**
-   **pb->show(); // base**
- }
- All the function calls here are statically bound

POINTERS TO DERIVED CLASSES (CONTD.)

- While it is permissible for a base class pointer to point to a derived object, the reverse is not true.
  - base b1;
  - derived *pd = &b1; // compiler error
- We can perform a **downcast** with the help of type-casting, but should use it with caution (see next slide).

POINTERS TO DERIVED CLASSES (CONTD.)

- Let we have –
  - class base { … };
  - class derived : public base { … };
  - class xyz { … }; // having no relation with “base” or “derived”
- Then if we write –
  - base b_obj; base *pb; derived d_obj; pb = &d_obj; // ok
  - derived *pd = pb; // compiler error
  - derived *pd = (derived *)pb; // ok, valid downcasting
  - xyz obj; // ok
  - pd = (derived *)&obj; // invalid casting, no compiler error, but may cause run-time error
  - pd = (derived *)&b_obj; // invalid casting, no compiler error, but may cause run-time error

POINTERS TO DERIVED CLASSES (CONTD.)

- In fact using type-casting, we can use pointer of any class to point to an object of any other class.
  - The compiler will not complain.
  - During run-time, the address assignment will also succeed.
  - But if we use the pointer to access any member, then it may cause run-time error.
- Java prevents such problems by throwing "ClassCastException" in case of invalid casting.

POINTERS TO DERIVED CLASSES (CONTD.)

- Pointer arithmetic is relative to the data type the pointer is declared as pointing to.
- If we point a base pointer to a derived object and then increment the pointer, it will not be pointing to the next derived object.
- It will be pointing to (what it thinks is) the next base object !!!
- **Be careful about this.**

IMPORTANT POINT ON INHERITANCE

- In C++, only public inheritance supports the perfect IS-A relationship.
- In case of private and protected inheritance, we cannot treat a derived class object in the same way as a base class object
  - Public members of the base class becomes private or protected in the derived class and hence cannot be accessed directly by others using derived class objects
- If we use private or protected inheritance, we cannot assign the address of a derived class object to a base class pointer directly.
  - We can use type-casting, but it makes the program logic and structure complicated.
- This is one of the reason for which Java only supports public inheritance.

INTRODUCTION TO VIRTUAL FUNCTIONS

- A virtual function is a member function that is declared within a base class and redefined (called ***overriding***) by a derived class.
- It implements the "one interface, multiple methods" philosophy that underlies polymorphism.
- The keyword **virtual** is used to designate a member function as virtual.
- Supports run-time polymorphism with the help of base class pointers.

INTRODUCTION TO VIRTUAL FUNCTIONS (CONTD.)

- While redefining a virtual function in a derived class, the function signature must match the original function present in the base class.
- So, we call it ***overriding***, not overloading.
- When a virtual function is redefined by a derived class, the keyword **virtual** is not needed (but can be specified if the programmer wants).
- The "virtual"-ity of the member function continues along the inheritance chain.
- A class that contains a virtual function is referred to as a ***polymorphic class***.

INTRODUCTION TO VIRTUAL FUNCTIONS (CONTD.)

- class base {
- public:
-   **virtual** void show() {
-     cout << “base\n”;
-   }
- };
- class derived : **public base** {
- public:
-   void show() {
-     cout << “derived\n”;
-   }
- };

- void main() {
-   base b1;
-   b1.show(); // base - (s.b.)
-   derived d1;
-   d1.show(); // derived – (s.b.)
-   base *pb = &b1;
-   pb->show(); // base - (d.b.)
-   **pb = &d1;**
-   **pb->show(); // derived (d.b.)**
- }
- Here,
  - s.b. = static binding
  - d.b. = dynamic binding

INTRODUCTION TO VIRTUAL FUNCTIONS (CONTD.)

```
class base {
public:
  virtual void show() {
    cout << "base\n";
  }
};
class d1 : public base {
public:
  void show() {
    cout << "derived-1\n";
  }
};
```

Run-time polymorphism

```
class d2 : public base {
public:
  void show() {
    cout << "derived-2\n";
  }
};
void main() {
  base *pb; d1 od1; d2 od2;
  int n;
  cin >> n;
  if (n % 2) pb = &od1;
  else pb = &od2;
  pb->show(); // guess what ??
}
```

VIRTUAL DESTRUCTORS

- Constructors cannot be virtual, but destructors can be virtual.
- It ensures that the derived class destructor is called when a base class pointer is used while deleting a dynamically created derived class object.

VIRTUAL DESTRUCTORS (CONTD.)

```
class base {
public:
   ~base() {
      cout <<  "destructing base\n";
   }
};
class derived : public base {
public:
   ~derived() {
      cout << "destructing derived\n";
   }
};
```

```
void main() {
   base *p = new derived;
   delete p;
}
```

- Output:
  - destructing base

Using non-virtual destructor

VIRTUAL DESTRUCTORS (CONTD.)

```
class base {
public:
  virtual ~base() {
    cout << "destructing base\n";
  }
};
class derived : public base {
public:
  ~derived() {
    cout << "destructing derived\n";
  }
};
```

```
void main() {
  base *p = new derived;
  delete p;
}
```

- Output:
  - destructing derived
  - destructing base

Using virtual destructor

MORE ABOUT VIRTUAL FUNCTIONS

- If we want to omit the body of a virtual function in a base class, we can use pure virtual functions.
  - virtual ret-type func-name(param-list) = 0;
- It makes a class an ***abstract class***.
  - We cannot create any objects of such classes.
- It forces derived classes to override it.
  - Otherwise they become abstract too.

MORE ABOUT VIRTUAL FUNCTIONS (CONTD.)

- Pure virtual function
  - Helps to guarantee that a derived class will provide its own redefinition.
- We can still create a pointer to an abstract class
  - Because it is at the heart of run-time polymorphism
- When a virtual function is inherited, so is its virtual nature.
- We can continue to override virtual functions along the inheritance hierarchy.

FINAL COMMENTS

- Run-time polymorphism is not automatically activated in C++.
- We have to use virtual functions and base class pointers to enforce and activate run-time polymorphism in C++.
- But, in Java, run-time polymorphism is automatically present as all **non-static methods** of a class are by default virtual in nature.
  - We just need to use superclass references to point to subclass objects to achieve run-time polymorphism in Java.

APPLYING POLYMORPHISM

- Early binding
  - Normal functions, overloaded functions
  - Nonvirtual member and friend functions
  - Resolved at compile time
  - Very efficient
  - But lacks flexibility
- Late binding
  - Virtual functions accessed via a base class pointer
  - Resolved at run-time
  - Quite flexible during run-time
  - But has run-time overhead; slows down program execution

LECTURE CONTENTS

- Teach Yourself C++
  - Chapter 10 (Full, with exercises)
  - Study the examples from the book carefully

MANIPULATING STRINGS

Resources

[1] Object Oriented Programming with C++ (3rd Edition) E Balagurusamy

[2] Teach Yourself C++ (3rd Edition) H Schildt

INTRODUCTION

- A string is a sequence of character.

- We have used null terminated <char> arrays (C-strings or C-style strings) to store and manipulate strings.

- ANSI C++ provides a class called **string**.

- We must include <string> in our program.

AVAILABLE OPERATIONS

- Creating string objects.
- Reading string objects from keyboard.
- Displaying string objects to the screen.
- Finding a substring from a string.
- Modifying string objects.
- Adding string objects.
- Accessing characters in a string.
- Obtaining the size of string.
- And many more.

COMMONLY USED STRING CONSTRUCTORS

- String();
  - // For creating an empty string.
- String(const char *str);
  - // For creating a string object from a null-terminated string.
- String(const string &str);
  - // For creating a string object from other string object.

CREATING STRING OBJECTS

- string s1, s3;	// Using constructor with no arguments.
- string s2(“xyz”);	// Using one-argument constructor.
- s1 = s2;	// Assigning string objects
- s3 = “abc” + s2;	// Concatenating strings

- cin >> s1;	// Reading from keyboard (one word)
- cout << s2;	// Display the content of s2
- getline(cin, s1)	// Reading from keyboard a line of text

- s3 += s1;	// s3 = s3 + s1;
- s3 += “abc”;	// s3 = s3 + “abc”;

MANIPULATING STRING OBJECTS

- string s1("12345");
- string s2("abcde");

- s1.insert(4, s2);   // s1 = 1234abcde5

- s1.erase(4, 5);   // s1 = 12345

- s2.replace(1, 3, s1);   // s2 = a12345e

MANIPULATING STRING OBJECTS

- insert()
- erase()
- replace()
- append()

RELATIONAL OPERATIONS

Operator	Meaning
==	Equality
!=	Inequality
<	Less than
<=	Less than or equal
>	Greater than
>=	Greater than or equal

- string s1("ABC"); string s2("XYZ");
- int x = s1.**compare**(s2);
  - x == 0 if s1 == s2
  - x > 0 if s1 > s2
  - x < 0 if s1 < s2

STRING CHARACTERISTICS

```
void display(string &str)
{
   cout << "Size = " << str.size() << endl;
   cout << "Length = " << str.length() << endl;
   cout << "Capacity = " << str.capacity() << endl;
   cout << "Max Size = " << str.max_size() << endl;
   cout << "Empty: " << (str.empty() ? "yes" : "no")
   << endl;
   cout << endl << endl;
}
```

STRING CHARACTERISTICS

Function	Task
size()	Number of elements currently stored
length()	Number of elements currently stored
capacity()	Total elements that can be stored
max_size()	Maximum size of a string object that a system can support
emply()	Return true or 1 if the string is empty otherwise returns false or 0
resize()	Used to resize a string object (effects only size and length)

ACCESSING CHARACTERS IN STRINGS

Function	Task
at()	For accessing individual characters
substr()	For retrieving a substring
find()	For finding a specific substring
find_first_of()	For finding the location of first occurrence of the specific character(s)
find_last_of()	For finding the location of first occurrence of the specific character(s)
[] operator	For accessing individual character. Makes the string object to look like an array.

COMPARING AND SWAPPING

- There is another overloaded version of compare

- int compare(int start_1, int length_1, string s_2, int start_2, int length_2)

- string s1, s2;
- int x = s1.compare(0, 2, s2, 2, 2);

- s1.swap(s2)
- Exchanges the content of string s1 and s2

LECTURE CONTENTS

- [1] Object Oriented Programming with C++ (3rd Edition) E Balagurusamy
  - Chapter 15 (Full)
- [2] Teach Yourself C++ (3rd Edition) H Schildt
  - Examples only
  - **Study the examples and exercise from both books carefully**