INHERITANCE – DERIVED CLASSES
Inheritance

- One of the Important Facets of OOD/OOP
- Allows to Create New Classes Based on Existing Ones – Reuse of Code
- Derived or Sub Classes From Base or Super Classes
- Indicates Dependence
- Single and Multiple Inheritance
Reasons For Inheritance

- Augmentation – Addition
- Diminution – Restriction – Rare and Inappropriate Inheritance Relation
- Specialization – Redefinition
Inheritance Vs. Containment

- Inheritance:
  - *Is-a Relationship*
  - E.g. A GM Car *is-a* Car

- Containment:
  - *Has-a Relationship*
  - E.g. A Car *has-a* Door
Base Class Member Access

- A Derived Class Can Access ONLY the Non-Private (Protected and Public) Members of the Base Class
- A Derived Class Can Further Restrict the Access to the Base Class Members
- A Derived Class Can Override Inherited Member Functions With Its Own Definitions
A Simple Derived Class – Example

class dummy{ //Base Class
    int pri_member;
    protected:
        int pro_member;
    public:
        int pub_member;
    dummy(){pri_member=0; pro_member=1; pub_member=2;}
    void all_access(){ //ALL access!
        cout << "Private: " << pri_member << endl;
        cout << "Protected: " << pro_member << endl;
        cout << "Public: " << pub_member << endl;}
};
class more_dummy: public dummy{ //Derived Class
    public:
        void restricted_access(){//Non-Private Member access
            cout << "Protected: " << pro_member << endl;
            cout << "Public: " << pub_member << endl;}
};
main(){ //ONLY Public Member Access
    dummy a_dummy_object; more_dummy a_more_dummy_object;
    a_dummy_object.all_access();
    a_dummy_object.pub_member = 9;
    a_dummy_object.all_access();
    a_more_dummy_object.restricted_access();
    a_more_dummy_object.pub_member = 99;
    a_more_dummy_object.all_access();}
A Simple Derived Class – Example

OUTPUT WILL BE

Private: 0
Protected: 1
Public: 2
Private: 0
Protected: 1
Public: 9
Protected: 1
Public: 2
Private: 0
Protected: 1
Public: 99
• Specifies the Class or Classes from Which the Class is Derived

```cpp
class rectangle{
  ....
};

class red{
  ....
};
/* Single Inheritance */
class square: public rectangle{
  ....
};
/* Multiple Inheritance */
class red_square: public square, public red{
  ....
};
```
Access Modifier

- Allows the Derived Class to Further Restrict the Access of Inherited Members of the Base Class
- `public`, `protected`, `private`
**public** Access Modifier – Example

- Inherited Private Members – No Access
- Inherited Protected Members – **protected** Access
- Inherited Public Members – **public** Access

```cpp
class base_class{
  int x;
  protected:
    int y;
  public:
    int z;
};
class derived_class: public base_class{
  public:
    void fn(){y = 0; z = 0;} //access "y" and "z".
};
main(){
  derived_class my_class;
  my_class.x++; //ERROR
  my_class.y++; //ERROR
  my_class.z++; //access to only "z".
}
```
**protected Access Modifier – Example**

- Inherited Private Members – No Access
- Inherited Protected Members – **protected** Access
- Inherited Public Members – **protected** Access

```cpp
class base_class{
  int x;
  protected:
    int y;
  public:
    int z;
};
class derived_class: protected base_class{
  public:
    void fn(){y = 0; z = 0;}//access "y" and "z".
};
class more_derived_class: public derived_class{
  /* Can access "y, z" and "fn()" */
      ....
};
main(){
  derived_class my_class;
  my_class.x++; //ERROR
  my_class.y++; //ERROR
  my_class.z++; //ERROR
}
```
**private Access Modifier – Example**

- Inherited Private Members – No Access
- Inherited Protected Members – `private` Access
- Inherited Public Members – `private` Access

```cpp
class base_class{
    int x;
    protected:
    int y;
    public:
    int z;
};
class derived_class: private base_class{
    public:
    void fn(){y = 0; z = 0;} // access "y" and "z".
};
class more_derived_class: public derived_class{
    /* Cannot access "x, y, z", but can access "fn()" */
    ....
};
main(){
    derived_class my_class;
    my_class.x++; // ERROR
    my_class.y++; // ERROR
    my_class.z++; // ERROR
}
```
virtual Base Class Specifier

- Ensures that Multiple Copies of Classes are NOT Present in the Derived Class

```cpp
class programming_language{
    /* General Features of Programming language */
    ....
};
class C: virtual public programming_language{
    /* Extra Features of C */
    ....
};
class Simula: virtual public programming_language{
    /* Extra Features of Simula */
    ....
};
class Cplusplus: public C, public Simula{
    /* Extra Features of C++ */
    ....
};
```
• Concept Similar to Constructor Initializer for a Class with Objects of Other Classes as Data Members

• Constructors of the Base Classes are Invoked (in a Sequence in which they are Specified) Before the Constructor for the Derived Class is Invoked

• If the Base Class itself is a Derived Class, the Process is Repeated Recursively

• If Base Class Constructor Requires Arguments, These are Supplied from the Inheritance Constructor

• Constructors are NOT Inherited, Although Default and Copy Constructors are Generated if Required
class programming_language{
    public:
        programming_language()
            {cout << "Programming is Fun!" << endl;}
    }

class C: virtual public programming_language{
    public:
        C(){cout << "C is a NICE Language!" << endl;}
    }

class Simula: virtual public programming_language{
    public:
        Simula(){cout << "Simula Had Classes!" << endl;}
    }

class C_plusplus: public C, public Simula{
    public:
        C_plusplus()
            {cout << "C++ is a NICE Languages and has Classes!" 
                << endl;}
    }

test{
    C_plusplus our_language;
}
Inheritance Constructor – Example

Default Constructors

OUTPUT WILL BE:
----- ---- ---
Programming is Fun!
C is a NICE Language!
Simula Had Classes!
C++ is a NICE Languages and has Classes!

If "virtual" is omitted from the class definitions of "C" and "Simula", then,

OUTPUT WILL BE:
----- ---- ---
Programming is Fun!
C is a NICE Language!
Programming is Fun!
Simula Had Classes!
C++ is a NICE Languages and has Classes!
#define MAX_SIZE 100

class car{ //Base class
    float engine_capacity;
    int no_of_doors, horse_power;
public:
    car (float cap, int door, int power) {
        engine_capacity = cap; no_of_doors = door;
        horse_power = power;
        cout << "Inside the Car Class!" << endl;
    }
};
class GM_car: public car{ //Derived Class (from "car")
    char *auto_maker;
public:
    GM_car(float cap, int doors, int hp)
    :car(cap, doors, hp) {
        auto_maker = new (char[MAX_SIZE]);
        strcpy(auto_maker, "GM");
        cout << "Inside the GM_car Class!" << endl;
    }
    ~GM_car() {
        {delete [] auto_maker;}
    }
};
main(){
    car simple_car(2.0, 2, 100);
    GM_car my_GM_car(2.5, 4, 150);
};
Inheritance Constructor – Example

Constructors With Arguments – Initialization

OUTPUT WILL BE:
------ ---- ---
Inside the Car Class!
Inside the Car Class!
Inside the GM_car Class!
Inheritance Destructors

- Destructor for the Derived Class is Invoked First
- Destructors for the Base Classes are Invoked in the Reverse Order, in which they are Specified
- Destructors are NOT Inherited, Although a Default Destructor is Generated if Required
Inheritance Destructors – Example

class programming_language{
    public:
    ~programming_language()
    {cout << "Programming is Fun!" << endl;}
};
class C: virtual public programming_language{
    public:
    ~C(){cout << "C is a NICE Language!" << endl;}
};
class Simula: virtual public programming_language{
    public:
    ~Simula(){cout << "Simula Had Classes!" << endl;}
};
class C_plusplus: public C, public Simula{
    public:
    ~C_plusplus()
    {cout << "C++ is a NICE Languages and has Classes!" << endl;}
};
main(){
    C_plusplus our_language;
}
Inheritance Destructors

OUTPUT WILL BE:
------ ---- ---
C++ is a NICE Languages and has Classes!
Simula Had Classses!
C is a NICE Language!
Programming is Fun!

If "virtual" is omitted from the class definitions of "C" and "Simula", then,

OUTPUT WILL BE:
------ ---- ---
C++ is a NICE Languages and has Classes!
Simula Had Classses!
Programming is Fun!
C is a NICE Language!
Programming is Fun!
**Virtual Destructors**

- Associated with Run-time Class of an Object Rather than Compile-time Class
- Safe and Wise to Make the Destructor of a Base Class as `virtual`

```cpp
class C{
    public:
        C(){cout << "In C!" << endl;}
        ~C(){cout << "Out-of C!" << endl;}
};

class C::public C{
    public:
        C::plusplus(){cout << "In C++!" << endl;}
        ~C::plusplus(){cout << "Out-of C++!" << endl;}
};

main(){
    C *ptr = new C::plusplus; //Polymorphism
    delete ptr;
}

OUTPUT WILL BE:
------- ---- --
In C!
In C++!
Out-of C!
Virtual Destructors – Example

class C{
    public:
    C(){cout << "In C!" << endl;}
    //"virtual" Destructor
    virtual ~C(){cout << "Out-of C!" << endl;}
};

class C_plusplus:public C{
    public:
    C_plusplus(){cout << "In C++!" << endl;}
    ~C_plusplus(){cout << "Out-of C++!" << endl;}
};

main(){
    C *ptr = new C_plusplus; //Polymorphism
    delete ptr;
}

OUTPUT WILL BE:
------ ---- --
In C!
In C++!
Out-of C++!
Out-of C!
INHERITANCE – CONT’D
Dominance

- When a Name Exists in More than One Class and that Class is Derived from a Class that Contains the Name, there is NO Ambiguity Present and the Entry in the Base Class Dominates.

```cpp
class A {
    public:
    void fn() {
        cout << "In the class A!" << endl;
    }
};
class B: public A {
    public:
    void fn() {
        cout << "In the class B!" << endl;
    }
};
class C: public B {
    public:
    C() {};
};
main() {
    C c_object;
    //Which "fn()" is Called???
    c_object.fn();
}
```

OUTPUT WILL BE:

------- ---- ---
In the class B!
virtual Member Functions

- Redefinition of the Base Class Member Functions in Derived Classes
- Indicate Specialization
- Compiler and the Loader Guarantee the Correct Correspondence Between Objects and the Functions Applied to them
- A **virtual** Function MUST be Defined in the Class in which it is First Declared – Exception Being **Pure Virtual Functions**
- If a Derived Class DOES NOT Need a Special Version of a **virtual** Function, it Need NOT Provide One
- A Function Defined **virtual** in a Base Class is **Implicitly Virtual** in Derived Classes
virtual Member Functions – Example

#define MAX 10
class rectangle{ //Base Class
protected:
    int side1, side2;
public:
    rectangle(int s1, int s2){side1 = s1; side2 = s2;}
    // Virtual Functions
    virtual int area(){return (side1*side2);}
    virtual int perimeter(){return (2*(side1+side2));}
};
class square: private rectangle{ //Derived Class
public:
    square(int s1, int s2):rectangle(s1, s2){
        if (side1 != side2)
            {cout << "Error–Unequal Sides!" << endl; exit(-1);}}
    int area(){return (side1*side1);} //Redefinition
    int perimeter(){return (4*side1);} //Redefinition
};
class red_square: private square{ //Derived Class
    char *color;
public:
    red_square(int s1, int s2):square(s1, s2)
    {color = new char [MAX]; strcpy(color, "Red");}
    virtual ~red_square(){delete [] color;}
    void get_color(){cout << "Color: " << color << endl;}
};
virtual Functions – Example – Cont’d

main()
{

rectangle one(10, 20);
cout << "Area of Rectangle: "
    << one.area() << endl;
cout << "Perimeter of Rectangle: "
    << one.perimeter() << endl;
square two(10, 10);
cout << "Area of Square: "
    << two.area() << endl;
cout << "Perimeter of Square: "
    << two.perimeter() << endl;
red_square three(20, 20);
cout << "Area of Red Square: "
    << three.area() << endl;
three.get_color();
}

OUTPUT WILL BE:

------- ---- ---
Area of Rectangle: 200
Perimeter of Rectangle: 60
Area of Square: 100
Perimeter of Square: 40
Area of Red Square: 400
Color: Red
Pure Virtual Functions and Abstract Classes

- A Pure Virtual Function is a `virtual` Function with a `Pure Specifier` (= 0)
- A Class with One or More Pure Virtual Functions is Called a Abstract Class
- NO INSTANCES (or OBJECTS) of an Abstract Class Can be Created, But It can be Used as a Base Class for Derived Classes
- Can be Used to Depict an Abstract Concept
class shape{ //Abstract Class
  public:
    virtual float area() = 0; //Pure Virtual Function
    virtual float perimeter() = 0; //Pure Virtual Function
};
class rectangle: public shape{ //Derived Class
  float side1, side2;
  public:
    rectangle(int s1, int s2){side1 = s1; side2 = s2;}
    float area(){return (side1*side2);}
    float perimeter(){return 2*(side1+side2);}
};
class circle: public shape{ //Derived Class
  float radius;
  public:
    circle(float s1){radius = s1;}
    float area(){return 3.1415 * radius * radius;}
    float perimeter(){return 2 * 3.1415 * radius;}
};
main()
{
  rectangle r1(10.0, 10.0);
  cout << "Area: " << r1.area() << endl;
  circle c1(12.34);
  cout << "Perimeter: " << c1.perimeter() << endl;
  shape dummy; // ERROR!!!!}
Multiple Inheritance

- A Class Derived From More than One Base Class
- Can Create Complex Class Declarations – Use With Caution
- Some OO Languages like Smalltalk DO NOT Support Multiple Inheritance
- Can Result Into Ambiguities – Use of :: Operator May be Necessary to Avoid the Ambiguities
Multiple Inheritance – Example

#define MAX 100
class square { //Base Class
    int side;
public:
square(int s) {side = s;}
    int area() {return side * side;}
};
class color { //Base Class
    char *color_name;
public:
color(char *ip)
    {color_name = new char[MAX], strcpy(color_name, ip);}
    virtual ~color() {delete [] color_name;}
    char *get_color() {return color_name;}
};
//Multiple Inheritance
class colored_square: public color, public square{
    public:
colored_square(int s, char *ip): square(s), color(ip){}
    void print_details()
    {
        cout << "Color: " << get_color() << endl;
        cout << "Area: " << area() << endl;
    }
};
main()
{
colored_square red_square(10, "Red");
    red_square.print_details();
}
Multiple Inheritance Ambiguity – Example

class square{ //Base Class
    int side;
    public:
    square(int s) {side = s;}
    void get_details()
    {cout << "Side: " << side << endl;}
};
class color{ //Base Class
    char *color_name;
    public:
    color(char *ip)
    {color_name=new char[MAX], strcpy(color_name, ip);}
    virtual ~color(){delete [] color_name;}
    void get_details()
    {cout << "Color: " << color_name << endl;}
};

//Multiple Inheritance
class colored_square: public color, public square{
    public:
    colored_square(int s, char *ip):square(s),color(ip){}
};
main()
{
    colored_square red_square(10, "Red");
    //AMBIGUITY...Which "get_details()" is Called????
    red_square.get_details();
}
Multiple Inheritance Ambiguity – Corrected Example

class square{ //Base Class
    int side;
    public:
        square(int s) {side = s;}
        void get_details(){cout << "Side: " << side << endl;}
    }

class color{ //Base Class
    char *color_name;
    public:
        color(char *ip)
        {color_name=new char[MAX],strcpy(color_name, ip);} // Virtual destructor prevents double delete
        virtual ~color(){delete [] color_name;}
        void get_details()
        {cout << "Color: " << color_name << endl;}
    }

    //Multiple Inheritance
    class colored_square: public color, public square{
        public:
            colored_square(int s,char *ip):square(s),color(ip){}
    }

    main(){ colored_square red_square(10, "Red");
        //square::get_details() -- Use of ::
        red_square.square::get_details();
        //color::get_details() -- Use of ::
        red_square.color::get_details();}
Polymorphism and Dynamic Binding

- Last Two Important Facets of OOD/OOP
- Polymorphism – Ability to Exists in More than One Form – A Polymorphic Reference is One that Can, Over Time, Refer to Different Instances of Different Classes
- Dynamic Binding – A Code Associated with a Function Call is Not Known Until the Run-time – Associated with Inheritance and Polymorphism
Polymorphism and Dynamic Binding – Example

class C_language{
    public:
        void ability()
            {cout << "C is a powerful language!" << endl;}
        virtual void nature()
            {cout << "C is an Imperative Language!" << endl;}
    }

class C_plusplus: public C_language{
    public:
        void ability()
            {cout << "C++ is a more powerful language!" << endl;}
        void nature()
            {cout << "C++ is a Language with Classes!" << endl;}
    }

main(){
    C_language first, *C_ptr = &first;
    C_plusplus second, *Cpp_ptr = &second;

    C_ptr->ability();     C_ptr->nature(); //static types
    Cpp_ptr->ability();   Cpp_ptr->nature(); //static types

    Cpp_ptr->C_language::ability(); //Use of ::
    Cpp_ptr->C_language::nature();  //Use of ::

    //Polymorphism and Dynamic Binding -- dynamic type
    C_ptr = &second;  C_ptr->ability();  C_ptr->nature();}
Polymorphism and Dynamic Binding – Example

OUTPUT WILL BE:
----- ---- ---
C is a powerful language!
C is an Imperative Language!
C++ is a more powerful language!
C++ is a Language with Classes!
C is a powerful language!
C is an Imperative Language!
C is a powerful language!
C++ is a Language with Classes!