OBJECT ORIENTED PROGRAMMING USING THE C++ LANGUAGE

BY

ISIDORO COUVERTIER REYES

AND

JEANNETTE SANTOS CORDERO

OBJECTIVES

AT THE END OF THIS TRAINING THE PARTICIPANTS WILL BE:

- **ABLE TO RECOGNIZE AND UNDERSTAND THE MOST FUNDAMENTAL OBJECT ORIENTED PROGRAMMING CONCEPTS.**
- **ABLE TO UNDERSTAND AND WRITE C++ LANGUAGE PROGRAMS.**
- **ABLE TO LEARN ON ITS OWN THE REST OF THE MORE ADVANCED FEATURES OF THE C++ LANGUAGE NOT COVERED HERE BECAUSE OF TIME CONSTRAINS.**

SOURCES

- ***TEACH YOURSELF C++*** **by H. SCHILDT, McGRAW HILL. THIS IS THE MAIN REFERENCE FOR THIS PRESENTATION. MOST OF THE EXAMPLES ARE FROM THIS BOOK.**

- ***C++ HOW TO PROGRAM*** **by H.M. DEITEL and P.J. DEITEL, PRENTICE HALL**

- ***A BOOK ON C*** **by A. KELLEY and I. POHL, ADDISON WESLEY**

OUTLINE

- **INTRODUCTION**
- **THE OBJECT ORIENTED PROGRAMMING PARADIGM**
- **C++ OVERVIEW**
- **CLASSES**
- **ARRAYS OF OBJECTS, POINTERS TO OBJECTS, AND REFERENCES**
- **OVERLOADING OF FUNCTIONS**
- **OVERLOADING OF OPERATORS**
- **INHERITANCE OF CLASSES**
- **VIRTUAL FUNCTIONS**

INTRODUCTION

- **INITIALLY MACHINE LANGUAGE PROGRAMMING, WAS O.K. FOR SMALL PROGRAMS.**
- **ASSEMBLY LANGUAGE ALLOWED WRITING LONGER PROGRAMS.**
- **HIGH LEVEL PROGRAMMING STARTED WITH FORTRAN, 1950s.**
- **STRUCTURED PROGRAMMING (CONTROL STRUCTURES, BLOCKS OF CODE, SUB-PROGRAMS, ETC.) IN THE 1960s TO ELIMINATE UNREADABLE CODE.**
- **OBJECT (CONTAIN INSTRUCTIONS AND DATA) ORIENTED TO REDUCE COMPLEXITY AND MANAGE LARGER PROGRAMS.**

OBJECT ORIENTED PROGRAMMING PARADIGM

- **ENCAPSULATION - CODE AND DATA BOUND TOGETHER SAFE FROM UNAUTHORIZED MANIPULATION (public and private).**

- **POLYMORPHISM - ONE NAME USED FOR MORE THAN ONE PURPOSE (FUNCTION OVERLOADING AND OPERATOR OVERLOADING). COMPILER DECIDES WHAT TO DO BASED ON AVAILABLE CONTEXT.**

- **INHERITANCE - AN OBJECT ACQUIRES (INHERITS) THE PROPERTIES OF ANOTHER OBJECT (CODE REUSE) AND ADDS ITS OWN PROPERTIES.**

C++ OVERVIEW - 1
A SIMPLE PROGRAM

```
#include <iostream.h>
//cin and cout stream manipulators
#define MAX 15
//C++ a superset of C in most aspects.

main()
{
//This is a comment
/* and this one too. */
        int i ;
        cout << "Enter a number: ";
        cin >> i;
        cout <<"You entered: "
        << i<< " and the maximum was: "
        << MAX;
        return 0;
//Could still use printf and scanf.
}
```

C++ OVERVIEW - 2 CLASSES

SYNTAX OF A CLASS DECLARATION

class ***name_of_the-class***
{ ***private functions and variables***
public:
public functions and variables
}*optional object_list;*

•PRIVATE FUNCTIONS AND VARIABLES CAN ONLY BE ACCESSED THROUGH MEMBER FUNCTIONS OF THE CLASS AND ARE HIDDEN OTHERWISE.

•PUBLIC FUNCTIONS AND VARIABLES HAVE NO ACCESS RESTRICTIONS AND ARE ACCESSED USING THE MEMBER ACCESS OPERATOR.

C++ OVERVIEW - 3 CLASSES

EXAMPLE

```
#include <iostream.h>

class example
{    int a, b, c; // private variables
public: // two public member functions
     void loadvars (int x, int y, int z);
     void print(){cout<<a<<"  "<<b<<"  “<<
                         c;} //in-lined?};

void example ::loadvars (int x, int y, int z)
{    a = x; b = y; c = z;}
// scope resolution operator ::
main()
{    example o1, o2;//object instantiation
      o1.loadvars(1, 2, 3);
      o1.print();
      return 0;}
```

C++ OVERVIEW - 4 CLASSES

•CLASS DECLARATION IS LIKE A NEW TYPE AND NO SPACE IS ALLOCATED UNTIL AN OBJECT IS CREATED (OR INSTANTIATED) IN MUCH THE SAME WAY THAT int AND float OCCUPY NO SPACE UNTIL A VARIABLE OF THAT TYPE IS DECLARED.

•OBJECTS o1 AND o2 ARE TWO DIFFERENT INSTANCES OF THE SAME CLASS.

•PUBLIC MEMBER FUNCTIONS loadvars() and print() BOTH HAVE ACCESS TO PRIVATE VARIABLES.

•WHAT IF a = 1 OR o1.a = 1 IN main()?

C++ OVERVIEW - 5 CLASSES

- **AN ATTEMPT TO DIRECTLY ACCESS ANY OF THE PRIVATE VARIABLES FROM OUTSIDE THE CLASS WILL GENERATE A COMPILE TIME ERROR. THUS a = 1 AND o1.a = 1 IN main() ARE BOTH ILLEGAL.**

C++ OVERVIEW - 6 CLASSES

EXAMPLE

```
#include <iostream.h>

class example
{   int a, b, c; // private variables
public:
    int d, e, f;
};

main()
{   example o1, o2;//object instantiation
    o2.d = 4;
    o2.e = 5;
    o2.f = 6;
    cout<<o2.d<< " "<<o2.e <<" "<<o2.f;
    return 0;
}
```

C++ OVERVIEW - 7 CLASSES

NOTE THAT PUBLIC VARIABLES d, e, AND f ARE ACCESSIBLE FROM WITHIN main() WITHOUT HAVING TO USE A MEMBER FUNCTION AS OPPOSSED TO private VARIABLES a, b, AND c.

C++ OVERVIEW - 8 FIRST DIFFERENCES WITH C

•AS SHOWN IN THE EXAMPLES, IN C++ THE USE OF void IN THE PARAMETER LIST OF A PARAMETERLESS FUNCTION IS OPTIONAL. WHAT DOES AN EMPTY PARAMETER LIST MEANS IN C?

•ALL FUNCTIONS MUST BE PROTOTYPED IN C++. A PROTOTYPE OF A MEMBER FUNCTION WITHIN A CLASS IS ALL THE PROTOTYPE IT NEEDS. IN C A FUNCTION PROTOTYPE IS OPTIONAL.

•IN C++ LOCAL VARIABLES CAN BE DECLARED ANYWHERE WHEREAS IN C THEY MUST BE DECLARED AT THE START OF A BLOCK.

C++ OVERVIEW - 9 POLYMORPHISM

EXAMPLE

```
#include <iostream.h>
int abs (int x)
{    if (x < 0)    return -x;
     else    return x;}

double abs (double x)
{    return x < 0 ? -x : x;}

long abs (long x)
{    return x < 0 ? -x : x;}

main()
{    int i = 3; double d= 3.14; long l = 4;
     cout << abs(i) << " " << abs(d) << " "
     << abs(l);
     return 0;}
```

C++ OVERVIEW - 10 POLYMORPHISM

OR

```
#include <iostream.h>
int abs (int x); // will not compile without these
double abs (double x);
long abs (long x);

main()
{    int i = 3; double d= 3.14; long l = 4;
     cout << abs(i) << " " << abs(d) << " "
     << abs(l);
     return 0;}

int abs (int x)
{    if (x < 0)       return -x;
     else    return x;}

double abs (double x)    {return x < 0 ? -x : x;}

long abs (long x)    {return x < 0 ? -x : x;}
```

CLASSES OVERV. - 1 CONSTRUCTORS AND DESTRUCTORS

•IN ORDER TO HAVE THE PROGRAM AUTOMATICALLY INITIALIZE AN OBJECT THE CONSTRUCTOR FUNCTION IS USED AS PART OF THE CLASS DECLARATION.

•A CLASS'S CONSTRUCTOR IS CALLED EACH TIME AN OBJECT OF THE CLASS IS CREATED.

•THE CONSTRUCTOR HAS THE SAME NAME AS THE CLASS AN IS NOT ALLOWED TO HAVE A RETURN TYPE.

•THE DESTRUCTOR FUNCTION (WHICH HAS THE SAME NAME AS THE CLASS BUT PRECEDED BY ~) SHOULD BE USED IF WE WANT SOME ACTIONS TO BE PERFORMED WHEN AN OBJECT IS DESTROYED.

CLASSES OVERV. - 2 CONSTRUCTORS AND DESTRUCTORS

EXAMPLE

```
#include <iostream.h>
class exclass
{    int a;
public:
     exclass(); // constructor prototype
     ~exclass();// destructor prototype
     void print();
};

exclass::exclass() // constructor function
{    cout << "Executing constructor!" << endl;
      a = 3;}

exclass::~exclass() // destructor function
{    cout << "Executing destructor!" << endl;}

void exclass::print() {cout << a << '\n';}

main()
{    exclass o1;
      o1.print();
      return 0;}
```

CLASSES OVERV. - 3 CONSTRUCTORS AND DESTRUCTORS

EXAMPLE

```
#include <iostream.h>, <assert.h>, <stdlib.h>
class mem
{    char *s;
public:
     mem();
     ~mem() { cout << "destructing\n"; free s;}  //delete
     void load();
};

mem::mem() // constructor function
{cout << "constructing!" << endl;
  s=(char *)malloc(sizeof(char [30])); //better new and delete
  assert(s != 0);} //macro calls abort in stdlib.h if needed

void mem::load()
{    cout << "Enter your lastname: ";
      cin >> s;}

main()
{    mem o1;
      o1.load();
      return 0;}
```

CLASSES OVERV. - 4 CONSTRUCTORS AND DESTRUCTORS

NOTES FOR THE PREVIOUS EXAMPLE:

•CONSTRUCTOR FUNCTION WAS USED TO INITIALIZE A POINTER TO A DYNAMICALLY ALLOCATED ARRAY OF CHARACTERS.

•DESTRUCTOR FUNCTION IS USED TO RELEASE THE DYNAMICALLY ALLOCATED MEMORY WHEN OBJECT IS DESTRUCTED.

•THOUGH malloc() AND free() ARE USED HERE, C++ PROVIDES new AND delete OPERATORS WHICH WE WILL USE IN THE FUTURE.

CLASSES OVERV. - 5 CONSTRUCTORS AND DESTRUCTORS

- **USE assert() OR SOMETHING LIKE:**

```
if (!s)
{   cout << “Allocation error\n”;
 exit(1);}
```

CLASSES OVERV. - 6 CONSTRUCTORS AND DESTRUCTORS

•CONSTRUCTORS MAY ALSO TAKE PARAMETERS.

```
#include <iostream.h>, <assert.h>, <stdlib.h>
class mem
{    int *s, num;
public:
    int i;
    mem(int size);
    ~mem() {cout << "destructing!\n"; free(s);}  //delete
    void init() {for(i=0;i<num;i++){s[i]=i; cout<<s[i]<<'\n';}}
};
mem::mem(int size) // constructor function
{    cout << "constructing!" << endl;
     num = size;
     s = (int *)malloc(sizeof(int)* size); // new and delete
     //s = new int [size];
     assert(s != 0);} //macro calls abort in stdlib.h if needed
main()
{    int n;
     cout << "Enter the size of the desired array: ";
     cin >> n;
     mem o1(n); //object instantiation
     o1.init();
     return 0;}
```

CLASSES OVERV. - 7 INHERITANCE

```
//first define the base class
class base
{
    int a;
public:
    void assign_a(int x) {a = x;}
    int value_of_a() {return a;}
};

//then define the derived class
class derived : public base
{
    int b;
public:
     void assign_b(int y) {b = y;}
     void aplusb() {cout << b + value_of_a();}
};
```

CLASSES OVERV. - 8 INHERITANCE

•THE BASE CLASS IS THE "PARENT" OF THE DERIVED "CHILD" CLASS.

•THE DERIVED CLASS CAN INHERIT THE BASE CLASS IN ONE OF SEVERAL WAYS, I.E. AS public, private, OR AS protected.

•AS WE WILL SEE LATER A DERIVED CLASS CAN INHERIT MORE THAN ONE BASE CLASS.

•A DERIVED CLASS CAN ADD ITS OWN VARIABLES AND MEMBER FUNCTIONS TO THOSE INHERITED FROM THE BASE CLASS.

CLASSES OVERV. - 9 INHERITANCE

•A DERIVED CLASS CAN NOT ACCESS WHAT IS private IN ITS BASE CLASS EXCEPT THROUGH THE BASE CLASS'S MEMBER FUNCTIONS TO WHICH IT HAS ACCESS (FOR NOW AT LEAST).

•WHEN A BASE CLASS IS INHERITED AS public ITS public PARTS REMAIN public IN THE DERIVED CLASS BUT THE private PARTS REMAIN private.

•WHEN A CLASS IS INHERITED AS protected ITS public AND protected MEMBERS ARE INHERITED AS protected.

CLASSES OVERV. - 10 INHERITANCE

- **THE protected SPECIFIER IS EQUIVALENT TO private AND IS USED TO KEEP A MEMBER private AND AT THE SAME TIME ALLOW A DERIVED CLASS ACCESS TO IT.**

- **protected MEMBERS ARE ACCESSIBLE FROM ANY DERIVED CLASS BUT ARE NOT ACCESSIBLE FROM OUTSIDE THE BASE OR DERIVED CLASSES.**

- **WHEN A BASE CLASS IS INHERITED AS private ALL ITS PARTS WILL BE private IN THE DERIVED CLASS.**

CLASSES OVERV. - 11 OBJECT POINTERS

```
#include <iostream.h>
class example
{    int a, b, c;
public:
     int w;
     void loadvars (int x, int y, int z);
     void print () {cout << a <<“ “<< b<<“ “ << c
               <<“ “<< w;}
};
void example ::loadvars (int x, int y, int z)
{    a = x; b = y; c = z;}

main()
{    example o1, *p = &o1;  // p points to ob1
      p -> w = 10; // o1.w = 10;
      p -> loadvars(1, 2, 3); //o1.loadvars(1, 2, 3);
      p -> print(); //o1.print();
      return 0;}
```

CLASSES OVERV. - 12 struct AND union

•IN C++ BOTH STRUCTURES AND UNIONS ARE EXPANDED TO INCLUDE BOTH DATA AND CODE JUST LIKE CLASSES.

•THE ONLY DIFFERENCE BETWEEN struct AND class IS THAT INSIDE struct THE KEYWORD private MUST BE USED TO SPECIFY THE PRIVATE PARTS SINCE BY DEFAULT EVERYTHING IS public.

•INSIDE union THE DEFAULT IS ALSO public. SOME COMPILERS DO NOT ALLOW UNIONS TO HAVE PRIVATE PARTS.

CLASSES OVERV. - 13 struct AND union

- **IN A union ALL DATA MEMBERS SHARE THE SAME MEMORY LOCATION.**

- **BOTH STRUCTURES AND UNIONS CAN HAVE CONSTRUCTORS AND DESTRUCTORS.**

CLASSES OVERV. - 14
struct AND union

```
#include <iostream.h>
struct example
{    int w;
     void loadvars (int x, int y, int z);
     void print () {cout << a << " "<< b <<" "<< c
          <<" "<< w;}
private:
     int a, b, c;
};
void example ::loadvars (int x, int y, int z)
{    a = x; b = y; c = z;}

main()
{    example o1, *p = &o1;  // p points to ob1
      p -> w = 10; // o1.w = 10;
      p -> loadvars(1, 2, 3); //o1.loadvars(1, 2, 3);
      p -> print(); //o1.print();
      return 0;}
```

CLASSES OVERV. - 15 struct AND union

```
#include <iostream.h>

union example
{   int i;
    float f;
    void print_it (int i) {cout << i << endl;}
    void print_it (float f) {cout << f << '\n';}
    // no private parts were used
};

main()
{    example o1;
     o1.i = 10;
     o1.print_it(o1.i);
     o1.f = 10.1;
     o1.print_it(o1.f);
     return 0;
}
```

More on CLASSES - 1 ASSIGNING OBJECTS

•TO BE ASSIGNED OBJECTS MUST BE OF THE SAME TYPE NAME.

```
#include <iostream.h>

class assign
{
     int a, b;
public:
     void load(int x, int y) {a = x; b = y;}
};

main()
{    assign o1, o2; //Must be of the same type
      o1.load(1, 2);// name in order to assign.
      o2 = o1; //Two different objects with
                    // the same values, bit-wise copy.
      return 0;}
```

More on CLASSES - 2 ASSIGNING OBJECTS

```
#include <string.h>
#include <assert.h>
class dyna
{       char *p;
        int l;
public:
        dyna(char *p1, int length, int which);
        ~dyna(){cout<<"destructing " << l
        << '\n';free(p);}
};

dyna ::dyna(char *p1, int length, int which)
{   p = (char *)malloc(sizeof(char)*length);
                l = which;
                assert(p != 0);
                strcpy (p, p1);
}
```

More on CLASSES - 3 ASSIGNING OBJECTS

```
main()
{       char *s1="Hello", *s2="Haiti";
        int l1, l2, n = 1;
        l1 = strlen(s1) + 1;
        l2 = strlen(s2) + 1;
        dyna o1(s1, l1, n);
        n++;
        dyna o2(s2, l2, n);
        o1 = o2; //trouble
        return 0;
}
```

More on CLASSES - 4 ASSIGNING OBJECTS

•IN THE PREVIOUS EXAMPLE WE NOTE THAT WHEN AN OBJECT OF TYPE dyna IS DESTROYED THE MEMORY IT ALLOCATED TO THE STRING IS RELEASED.

•BECAUSE OF THE ASSIGNMENT OF o2 TO o1 WE FIND THAT p FROM o2 WILL POINT TO THE SAME LOCATION AS p FROM o1.

•WHAT HAPPENS WHEN THE OBJECTS ARE DESTROYED? MEMORY POINTED TO BY p FROM o1 IS RELEASED TWICE WHEREAS THAT FROM o2 IS NEVER RELEASED. THIS COULD CAUSE A CRASH.

More on CLASSES - 5 ASSIGNING OBJECTS

•THE PROBLEM ARISES BECAUSE THE OBJECT HAS A DESTRUCTOR THAT RELEASES MEMORY PREVIOUSLY ALLOCATED.

•MAKE SURE THAT WHEN OBJECTS ARE ASSIGNED YOU DO NOT DESTROY DATA THAT WILL BE USED LATER ON.

•THE CONSTRUCTOR FOR A GLOBAL OBJECT IS CALLED ONCE. FOR A LOCAL OBJECT IT IS CALLED EACH TIME THE DECLARATION IS REACHED. THE DESTRUCTOR IS CALLED EACH TIME THE OBJECT IS DESTROYED.

More on CLASSES - 6 PASSING OBJECTS

```
#include <iostream.h>

class pass_obj
{     int a;
public:
      pass_obj(int x) {  a = x;
                         cout <<“constructing\n”;}
      ~pass_obj() {cout << “destructing\n”;}
      int value_a() {return a;}
};

// declare parameter as an object
int double_it(pass_obj o)//default is by value
{     return 2*o.value_a();}

main()
{     pass_obj o1(3);
      cout << double_it(o1) << endl;
      return 0;}
```

More on CLASSES - 7 PASSING OBJECTS

•BY DEFAULT OBJECTS ARE PASSED BY VALUE AND THUS A COPY OF THE OBJECT IS MADE. CHANGING THE COPY DOES NOTHING TO THE ORIGINAL.

•WHEN MAKING A COPY OF AN OBJECT IN A FUNCTION CALL THE CONSTRUCTOR OF THE COPY IS NOT CALLED.

•IF A FUNCTION RECEIVED A COPY OF AN OBJECT AS A PARAMETER, THEN WHEN THE FUNCTION TERMINATES THE DESTRUCTOR FOR THE COPY IS CALLED.

More on CLASSES - 8 PASSING OBJECTS

•IF THE OBJECT THAT IS PASSED AS A PARAMETER ALLOCATES MEMORY DYNAMICALLY AND ITS DESTRUCTOR FREES IT, THEN WHEN THE FUNCTION TERMINATES THE COPY OF THE OBJECT WILL RELEASE THE SAME MEMORY LEAVING THE CALLING OBJECT INOPERATIVE.

•ONE WAY TO SOLVE THIS PROBLEM IS TO PASS THE ADDRESS (USING &) OF THE OBJECT IF THE CONDITIONS EXPLAINED ABOVE ARE MET. IN THE FUNCTION DECLARATION DECLARE THE FORMAL AS A POINTER TO AN OBJECT OF THE APPROPRIATE TYPE AND USE -> TO ACCESS ITS PARTS.

More on CLASSES - 9 RETURNING OBJECTS

•WHEN AN OBJECT IS RETURNED FROM A FUNCTION, AN INVISIBLE TEMPORARY OBJECT IS GENERATED. THE TEMPORARY OBJECT WILL HAVE ITS DESTRUCTOR CALLED SINCE IT IS IN EFFECT A COPY OF AN OBJECT.

• THE ABOVE MAY DESTROY THE DYNAMIC ALLOCATION SYSTEM DEPENDING ON THE COMPILER, MEMORY MODEL USED, ETC. AND IT SHOULD BE AVOIDED.

•NOTE THAT THIS ARISES WHEN THE DESTRUCTOR FREES DYNAMICALLY ALLOCATED MEMORY.

More on CLASSES - 10 RETURNING OBJECTS

```
#include <iostream.h>,<stdlib.h>,<assert.h>
class reto
{  int *a;
public:
   reto() {a=NULL;cout<<"constructing\n";}
   ~reto(){cout << "destructing\n";}
   void init(int *c) {a = c;}};
reto aloc()
{  reto t; int num, i, *p;
   cout << "Enter a positive integer: ";
   cin >> num;
   p = (int *)malloc(sizeof(int)*num);
   assert(p != 0);
   for(i = 0; i < num; i++)
   {    p[i] = rand(); cout << p[i] << '\n';}
   t.init(p);
   return t;} // invisible object generated!!!!
```

More on CLASSES - 11 RETURNING OBJECTS

```
main()
{
        reto o1; //Null pointer assignment
        o1 = aloc();
        return 0;
}
```

•THE DESTRUCTOR IS CALLED ONCE WHEN OBJECT t GOES OUT OF SCOPE, I.E. WHEN FUNCTION aloc() RETURNS (NOTE THAT t IS LOCAL TO aloc()). IT IS CALLED TWO MORE TIMES WHEN THE PROGRAM ENDS TO DESTROY THE INVISIBLE COPY OF t THAT WAS AUTOMATICALLY GENERATED WHEN THE OBJECT WAS RETURNED AND FOR AND TO DESTROY IT.

More on CLASSES - 12 FRIEND FUNCTIONS

- **FRIEND FUNCTIONS ARE USED TO ALLOW A FUNCTION ACCESS TO THE PRIVATE MEMBERS OF A CLASS WITHOUT BEING A MEMBER OF THE CLASS.**

- **WILL COME BACK TO MORE USES OF FRIENDS FUNCTIONS LATER (OPERATOR OVERLOADING).**

- **FOR NOW WILL CONCENTRATE ON ALLOWING A FUNCTION ACCESS TO TWO OR MORE CLASSES.**

More on CLASSES - 13 FRIEND FUNCTIONS

```
#include <iostream.h>
class class1
{  int x, y;
public:
   class1 (int a, int b) {x = a; y = b;}
   friend int yfactx(class1 o);
};
int yfactx(class1 o)
{  if (o.x % o.y) return 0; return 1;}

main()
{  int t = 1, u = 5;
    char *s1 = " a factor of ", *s2 = " is not";
    class1 o1(t, u);
    yfactx(o1)? cout<<u<<s1<<t:
          cout << u << s2 << s1 << t;
    return 0;}
```

More on CLASSES - 14 FRIEND FUNCTIONS

•NOTE THAT IN THE PREVIOUS EXAMPLE IT WOULD BE WRONG TO CALL yfactx() IN THIS WAY o1.yfactx() BECAUSE yfactx() IS NOT A MEMBER OF THE CLASS.

•WHEN A MEMBER FUNCTION REFERS TO A PRIVATE ELEMENT, THE COMPILER KNOWS WHICH OBJECT THE ELEMENT BELONGS TO BY THE OBJECT LINKED TO THE FUNCTION WHEN THE MEMBER FUNCTION IS EXECUTED.

More on CLASSES - 15 FRIEND FUNCTIONS

- **WITHIN yfactx() IT WOULD BE WRONG TO REFER TO THE CLASS'S VARIABLES DIRECTLY BECAUSE OBJECT o1 IS NOT LINKED TO THE FRIEND FUNCTION BECAUSE FRIEND FUNCTIONS ARE NOT LINKED TO ANY OBJECT.**

More on CLASSES - 16 FRIEND FUNCTIONS

•MEMBER OF ONE CLASS, FRIEND OF ANOTHER

```
class orng;  //this is a forward reference

class grpfrt
{      int trees,  seeds;
public:
       grpfrt (int t, int s) { trees = t; seeds = s;}
       int numseeds_grter(orng c);
};

class orng
{     int weight,  seeds;
public:
      orng (int w, int s) { weight = w; seeds = s;}
      friend int grpfrt::numseeds_grter(orng c);
};
```

More on CLASSES - 17 FRIEND FUNCTIONS

```
//only objects of class orng need to be passed
//since numseeds_grter() is a member function
//of class grpfrt
int grpfrt::numseeds_grter(orng o)
{    return seeds - o.seeds;}

main()
{    orng o1( 3, 21);
     grpfrt o2(10, 24);
     int a;

//the call to numseeds_grter() is performed as
//a member of grpfrt
     a = o2.numseeds_grter(o1);
//     .
//     .
//     .
}
```

More on CLASSES - 18 FRIEND FUNCTIONS

•NOTE THE USE OF THE FORWARD REFERENCE SO THAT THE COMPILER WILL KNOW ABOUT orng BEFORE ITS USE IN grpfrt WHICH COMES BEFORE THE DECLARATION OF orng.

•NOTE THE USE OF THE SCOPE RESOLUTION OPERATOR :: WHEN THE FRIEND FUNCTION IS PROTOTYPED. THIS IS TO TELL THE COMPILER THAT THE FRIEND FUNCTION IS A MEMBER OF CLASS grpfrt.

ARRAYS, POINTERS, and REFERENCES - 1

•OBJECTS CAN BE ARRAYED

```
#include <iostream.h>
class apr
{     int a;
public:
        apr(int x) { a = x;}// single parameter
        void display() {cout << a << ‘  ‘;}
};

main()
{       int i, j ;
        apr o1[2][3] = {1, 2, 3, 4, 5, 6};
//     apr o1[2][3] ={apr(1), apr(2), …, apr(6)};
        for (i = 0; i < 2; i++)
            for (j = 0; j < 3; j++)
                o1[i][j].display();
        return 0;}
```

ARRAYS, POINTERS, and REFERENCES - 2

```
class apr
{     int a, b;
public:
      apr(int x, int y) { a = x; b = y;}
      void display() {cout << a << ‘ ‘ << b <<
                              ‘\n’;}};

main()
{      int i, j ;
// for constructors with two or more
// parameters must use this form
      apr o1[2][2] = {apr(1, 2), apr(3, 4),
                              apr(5, 6), apr(7, 8)};
      for (i = 0; i < 2; i++)
          for (j = 0; j < 3; j++)
              o1[i][j].display();
      return 0;}
```

ARRAYS, POINTERS, and REFERENCES - 3

•C++ PROVIDES ANOTHER WAY TO ALLOCATE AND RELEASE MEMORY DYNAMICALLY THROUGH new AND delete. THE SYNTAX IS AS SHOWN:

pointer_variable* = new *type;
delete *pointer_variable*;

•TO INITIALIZE A DYNAMICALLY ALLOCATED OBJECT:

***pointer_variable* = new *type* (*initial_value*);**

•TO DYNAMICALLY ALLOCATE A ONE-DIMENSIONAL ARRAY:

***pointer_variable* = new *type*[*size*];**
delete [*size*] *pointer_variable*;// NOTE

ARRAYS, POINTERS, and REFERENCES - 4

•NOTE: delete [*size*] *pointer_variable*; THIS FORM MAY BE NEEDED, E.G. WHEN A DYNAMICALLY ALLOCATED ARRAY OF OBJECTS IS TO BE RELEASED. IT CAUSES THE COMPILER TO CALL THE DESTRUCTOR FUNCTION FOR EACH ELEMENT. THE POINTER VARIABLE IS FREED ONCE. WITHOUT *size* THE DESTRUCTOR FUNCTION FOR EACH ELEMENT WILL NOT BE CALLED. IF THERE IS NO DESTRUCTOR, THEN THERE IS NO NEED FOR *size*.

•ARRAYS THAT ARE DYNAMICALLY ALLOCATED CAN NOT BE INITIALIZED AT THE MOMENT OF CREATION.

ARRAYS, POINTERS, and REFERENCES - 5

```
#include <iostream.h>,<stdlib.h>,<assert.h>

class nodest
{   int a, b;
public:
    void init(int x, int y) { a = x; b = y;}
//  ...
};
main()
{   nodest *p1;

    p1 = new nodest [5];  // array of objects
    assert( p1 != 0);
//  ...
    delete p1;  // use this form since no destr.
    return 0;
}
```

ARRAYS, POINTERS, and REFERENCES - 6

```
#include <iostream.h>,<stdlib.h>,<assert.h>

class class1
{   int a, b;
public:
    void init(int x, int y) { a = x; b = y;}
    ~class1() { cout << “destroying\n; // ...}
//  ...
};
main()
{   class1 *p1;

    p1 = new class1 [5];  // array of objects
    assert( p1 != 0);
//  ...
    delete [5] p1;
    return 0;}
```

ARRAYS, POINTERS, and REFERENCES - 7

•A REFERENCE IS AN IMPLICIT POINTER BUT IS USED IN A WAY DIFFERENT FROM A POINTER.

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

void swap(int &a, int &b)// with references
{     int temp;
      temp = a; // actual value is copied
      a = b;
      b = temp;
}
```

ARRAYS, POINTERS, and REFERENCES - 8

•THE CALL TO THE VERSION OF swap() THAT USES POINTERS IS SOMETHING LIKE:

swap(&i, &j);

•THE CALL TO THE VERSION OF swap() THAT USES REFERENCES IS SIMPLY:

swap(i, j);

BUT IT IS THE ACTUAL ADDRESSES OF BOTH i and j WHICH ARE PASSED. INSIDE swap() WHENEVER a AND b ARE USED IT IS THE ACTUAL VALUES, NOT COPIES, OF i AND j WHICH ARE EITHER READ OR REPLACED.

ARRAYS, POINTERS, and REFERENCES - 9

•USING REFERENCES IS A NICE WAY TO SOLVE THE PROBLEM WE FACE WHENEVER AN OBJECT IS PASSED BY VALUE. REMEMBER THAT WHEN AN OBJECT IS PASSED BY VALUE TO A FUNCTION THE CONSTRUCTOR IS NOT CALLED BUT THE OBJECT'S DESTRUCTOR IS CALLED. JUST USE THE & OPERATOR IN THE FUNCTION DECLARATION AND PROTOTYPE AND JUST THE NAME OF THE FORMAL IN THE FUNCTION'S BODY. DO NOT USE THE -> MEMBER ACCESS OPERATOR BUT DO USE THE . ACCESS OPERATOR WHEN USING REFERENCES TO AN OBJECT.

ARRAYS, POINTERS, and REFERENCES - 10

```
class class1
{   int a, b;
public:
    void init(int x, int y) { a = x; b = y;}
    ~class1() { cout << “destroying\n; // ...}
//   ...
};

void f1(class1 o) //by value
{   o.init(2, 3);} //will invoke destructor when
//returning and when program terminates

vs.

void f1(class1 &o) //by reference
{    o.init(2, 3);}  //the destructor will not be
//invoked when returning but note use of .
```

ARRAYS, POINTERS, and REFERENCES - 11

•REFERENCES CAN ALSO BE RETURNED BUT DO BE CAREFUL.

```
int &f1()
{     return x;}//makes sense if x is a global

but

int &f1()
{     int x;
       return x;}//makes no sense since x,
//being local to f1() will go out of scope when
//the function returns.
```

ARRAYS, POINTERS, and REFERENCES - 12

•IMPLEMENTING ARRAY BOUNDS CHECKING

```
#include <assert.h>
class  class1
{     int *p, size;
public:
   class1(int s) { size = s; p = new int [size];
                        assert(p != 0);}
   ~class1() {delete [size] p;}
   int &write_to_array(int i);
//...
};
int &class1::write_to_array(int i)
{     assert(i >= 0 && i < size);
       return p[i];// this returns a reference to
                        // element p[i]
}
```

ARRAYS, POINTERS, and REFERENCES - 13

```
main()
{    class1 o1(30);

     o1.write_to_array(2) = 7; // p[2] = 7;
//   ...
     return 0;
}
```

FUNCTION OVERLOADING - 1

•OVERLOADING CONSTRUCTORS

```
class overl
{    int a;
public:
      overl() { a = 0;} //this takes no parameter
      overl(int x) {a = x;} //this constructor
//                                  takes 1 parameter
};

main()
{     overl o1, o2(5);
      overl o3[4], o4[6] = {1, 2, 3, 3, 2, 1};
//...
       return 0;}
```

FUNCTION OVERLOADING - 2

- **USING DEFAULT ARGUMENTS**

```
class def
{     int a;
public:
      def(int x = 0) {a = x;}
};

main()
{     def o1;//will use default value
      def o2(3);//will initialize to 3
//    ...
      return 0;
}
```

FUNCTION OVERLOADING - 3

- **AMBIGUITY 1**

```
float f1(float a, float b)
{    return a + b;}

double f1(double a, double b)
{    return a * b;}
```

ASSUME THE FOLLOWING:

```
float x = 3.145, y = 0.1;
double t = 10.0, u = 111.2345;

cout << f1(x, y); //this is ok
cout << f1(t, u); //this is ok too
cout<< f1(2.0, 3.1); //this is ambiguous even
//though the functions are not ambiguous
```

FUNCTION OVERLOADING - 4

- **AMBIGUITY 2**

```
void  f1(float a)
{    cout << a;}

void f1(float &a)
{    a =  a + 1.1;
     cout << a;
}
```

ASSUME THE FOLLOWING:

```
float x = 3.145;

f1(x); //this is ambiguous
```

FUNCTION OVERLOADING - 5

- **AMBIGUITY 3**

```
float f1(float a)
{    return a + 1.1;}

float f1(float  a, float b = 0)//  b def.  Par.
{    return a * b;}
```

ASSUME

```
cout << f1(3.2);//this is ambiguous
cout << f1(2.4, 5.1);//this is unambiguous
```

OPERATOR OVERLOADING - 1

•OPERATORS CAN BE OVERLOADED BY CREATING AN OPERATOR FUNCTION (MEMBER OR FRIEND). THE OVERLOADING IS ALWAYS RELATIVE TO A CLASS.

•THE TEMPLATE OR SYNTAX FOR A MEMBER OPERATOR FUNCTION IS:

***ret_type class_name*::operator#(*list_of_args*)**
{ //... }

WHERE THE OVERLOADED OPERATOR IS TO BE SUBSTITUTED FOR THE # SYMBOL. THE RETURN TYPE IS USUALLY THE CLASS FOR WHICH IT IS DEFINED.

OPERATOR OVERLOADING - 2

•THE PRECEDENCE OF AN OPERATOR AND THE NUMBER OF OPERANDS IT TAKES CANNOT BE ALTERED BY OVERLOADING IT.

•THE FOLLOWING OPERATORS CANNOT BE OVERLOADED:

. : .* ?:

•THE PREPROCESSOR OPERATORS CANNOT BE OVERLOADED (# AND ##).

•OPERATOR FUNCTIONS ARE INHERITED EXCEPT FOR =.

•OPERATOR FUNCTIONS CANNOT HAVE DEFAULT PARAMETERS.

OPERATOR OVERLOADING - 3

```
#include <iostream.h>
class coord
{    int x, y, z;
public:
     coord() {x = 0; y = 0; z = 0;}
     coord(int i, int j, int k) {x = i; y = j; z = k;}
     friend void display(coord o);
     coord operator+ (coord o2);
};

coord coord::operator+ (coord o2)
{    coord t;
      t.x = x + o2.x;
      t.y = y + o2.y;
      t.z = z + o2.z;
      return t;//allows o3 = o1 + o2;
}
```

OPERATOR OVERLOADING - 4

```
void display(coord o)
{cout << o.x <<‘ ‘<< o.y <<‘ ‘<< o.z << ‘\n’;}

main()
{
    coord o1(1, 2, 3), o2(4, 5, 6), o3;

    display(o3);
    o3 = o1 + o2;//o1 generates the call to
//                      operator+()
    display(o3);//display(o1 + o2);
    return 0;
}
```

OPERATOR OVERLOADING - 5

•IN STATEMENT o3 = o1 + o2; OBJECT o2 (THE OBJECT TO THE RIGHT OF THE +) IS PASSED AS AN ARGUMENT TO THE OPERATOR FUNCTION. THE OBJECT TO THE LEFT OF THE + SIGN (o1 IN THIS CASE) GENERATES THE CALL AND IS PASSED IMPLICITLY BY USING THE this POINTER. THIS IS TRUE FOR ANY OVERLOADED BINARY OPERATOR FUNCTION.

•THE this POINTER IS PASSED TO ANY MEMBER FUNCTION WHEN THE MEMBER FUNCTION IS CALLED AND IT IS A POINTER TO THE OBJECT THAT GENERATES THE CALL. THIS IS DONE AUTOMATICALLY.

OPERATOR OVERLOADING - 6

•SAY WE WANTED TO OVERLOAD THE = OPERATOR.

```
coord coord::operator= (coord o2)
{     x = o2.x;
      y = o2.y;
      z = o2.z;
      return *this;
}
```

•RETURNING *this ALLOWS THE CLASS'S OBJECTS TO BE USED IN STATEMENTS LIKE:

```
o1 = o2 = o3;
```

OPERATOR OVERLOADING - 7

•COULD ALSO OVERLOAD A BINARY OPERATOR USING A BUILT-IN TYPE ON THE RIGHT OF THE OPERATOR.

```
coord coord::operator+ (int i)
{     coord t;
      t.x = x + i;
      t.y = y + i;
      t.z = z + i;
      return t;
}
```

•JUST NEED TO MAKE SURE THE BUILT-IN TYPE IS INDEED ON THE RIGHT AT THE MOMENT OF THE CALL.

```
      o2 = o1 + 3;// ok
      o3 = 5 + o4;//an error to be solved using
//a friend rather than a member function
```

OPERATOR OVERLOADING - 8

•TO AVOID THE PROBLEMS ASSOCIATED WITH PASSING A COPY OF AN OBJECT USE REFERENCES:

```
coord coord::operator+ (coord &o2)
{    coord t;
     t.x = x + o2.x;
     t.y = y + o2.y;
     t.z = z + o2.z;
     return t;
}
```

•REMEMBER THAT THERE ARE ALSO PROBLEMS ASSOCIATED WITH RETURNING AN OBJECT WHEN THE OBJECT'S DESTRUCTOR FREES DYNAMICALLY ALLOCATED MEMORY.

OPERATOR OVERLOADING - 9

•AS OPPOSED TO THE OTHER BINARY OPERATORS THE RELATIONAL OPERATORS ARE NOT EXPECTED TO RETURN OBJECTS OF THE CLASS.

```
int coord::operator&& (coord o2)
{    int andx, andy, andz;
     andx = x && o2.x;
     andy = y && o2.y;
     andz = z && o2.z;
     return (andx && andy && andz);
}
```

OPERATOR OVERLOADING - 10

•WHEN OVERLOADING A UNARY OPERATOR USING A MEMBER FUNCTION THE FUNCTION HAS NO PARAMETERS. THE CALL TO THE OPERATOR FUNCTION IS GENERATED BY THE OPERAND.

```
coord coord::operator++ ()
{
    x++;
    y++;
    z++;
    return *this; //to allow o2 = o1++;, etc.
}//o1++;
```

OPERATOR OVERLOADING - 11

•TO DISTINGUISH BETWEEN PREFIX AND POSTFIX UNARY OPERATORS OVERLOAD TWICE (AS SHOWN PREVIOUSLY AND AS FOLLOWS).

```
coord coord::operator++ (int a)
{    //a receives 0
     ++x;
     ++y;
     ++z;
     return *this; //to allow o2 = ++o1;, etc.
}//++o1;
```

THIS MAY OR MAY NOT BE SUPPORTED BY THE COMPILER YOU USE.

OPERATOR OVERLOADING - 12

•YOU MAY WANT TO OVERLOAD THE MINUS SIGN TWICE SINCE IT IS BOTH BINARY AND UNARY.

```
coord coord::operator- (coord o2) //binary
{    coord t;
     t.x = x - o2.x;
     t.y = y - o2.y;
     t.z = z - o2.z;
     return t;
};

coord coord::operator- () //unary
{     x = -x;
      y = -y;
      z = -z;
      return *this;}
```

OPERATOR OVERLOADING - 13

•FRIENDS FUNCTIONS ARE NOT MEMBER FUNCTIONS AND ARE THUS NOT PASSED THE this POINTER. OPERANDS ARE PASSED EXPLICITLY (TWO FOR BINARY, ONE FOR UNARY).

•FRIEND FUNCTIONS CAN BE USED TO SOLVE THE PROBLEM OF PASSING A BUILT-IN TYPE THAT IS ON THE LEFT OF THE OPERATOR. JUST DEFINE TWO FRIEND FUNCTIONS (RATHER THAN ONE MEMBER FUNCTION): ONE TO COVER THE CASE WHERE THE BUILT-IN TYPE IS ON THE LEFT AND ONE FOR THE CASE WHEN THE BUILT-IN TYPE IS ON THE RIGHT.

OPERATOR OVERLOADING - 14

•FRIEND FUNCTIONS CANNOT BE USED TO OVERLOAD THE ASSIGNMENT OPERATOR. YOU CAN ONLY USE MEMBER FUNCTIONS IN THIS CASE.

•SINCE FRIEND FUNCTIONS ARE NOT PASSED THE this POINTER, THEN WHEN USED TO OVERLOAD THE ++ AND -- OPERATORS THE OPERAND MUST BE PASSED BY REFERENCE. THIS IS SUCH THAT THE OPERAND IS ACTUALLY MODIFIED. HOW DO YOU NOW DISTINGUISH BETWEEN PREFIX AND POSTFIX?

OPERATOR OVERLOADING - 15

•REMEMBER THAT THE ASSIGNMENT OPERATOR WHEN APPLIED TO AN OBJECT WILL RESULT IN A BIT-WISE COPY OF THE OBJECT. ASSUME THAT THE OBJECT HAS A DESTRUCTOR THAT FREES DYNAMICALLY ALLOCATED MEMORY.

IF A BIT-WISE COPY IS MADE, THEN A POINTER IN THE FIRST WILL BE COPIED ONTO THE SECOND (BOTH POINTERS POINT TO EXACTLY THE SAME PIECE OF MEMORY OR TO THE SAME FILE).

WHAT HAPPENS IF THE DESTRUCTOR FOR EITHER ONE IS CALLED? TO AVOID THIS SITUATION IT MAY BE NECESSARY TO HANDLE THE COPY PROCESS DIFFERENTLY BY OVERLOADING THE ASSIGNMENT OPERATOR.

INHERITANCE - 1

•TEMPLATE FOR DERIVING A CLASS FROM ANOTHER:

class *derived_name*:*access_spec base_name*
{ // *... derived class body* }

WHERE ACCESS IS ONE OF:

1. public - ALL public MEMBERS OF THE BASE CLASS ARE INHERITED AS public IN THE DERIVED CLASS BUT THE private MEMBERS REMAIN private. THE protected MEMBERS BECOME protected.

2. private - ALL MEMBERS (private, protected, AND public) OF THE BASE ARE INHERITED AS private.

INHERITANCE - 2

3. protected - WHEN A CLASS IS INHERITED AS protected ITS public AND protected MEMBERS ARE INHERITED AS protected.

•protected MEMBERS ARE ACCESSIBLE FROM ANY DERIVED CLASS BUT ARE NOT ACCESSIBLE FROM OUTSIDE THE BASE OR DERIVED CLASSES.

•REMEMBER THAT private MEMBERS WITHIN A CLASS REMAIN private REGARDLESS OF HOW IT (THE CLASS) IS INHERITED.

•protected AND private INHERITANCE ARE RARE. BE CAREFUL WITH BOTH.

INHERITANCE - 3

```
#include <iostream.h>

class base
{   int a;
public:
     void init_a(int x) {a = x;}
     void print_a() {cout << a << endl;}
};

class derived:public  base
{   int b;
public:
    void init_b(int y) {b = y;}
    void print_b() {cout << b << endl;}
};
```

INHERITANCE - 4

```
main()
{    derived o1;

     o1.init_b(1);
     o1.init_a(2);
     o1.print_b();
     o1.print_a();

//   o1.a = 3;an error since a is private to base
//   o1.b = 4;and so is b to derived````

     return 0;
}
```

INHERITANCE - 5

```
#include <iostream.h>

class base
{   int a;
public:
     void init_a(int x) {a = x;}
     void print_a() {cout << a << endl;}
};

class derived:private  base
{   int b;
public:
    void init_b(int y) {b = y;}
    void print_b() {cout << b << endl;}
};
```

INHERITANCE - 6

```
main()
{    derived o1;
     base o2;

     o1.init_b(1);
     o1.print_b();

//   o1.init_a(2);//error since inherited as
//                      private
//   o1.print_a();//error since inherited
//                      as private

      o2.init_a(2);//still accessible from base
      o2.print_a();

     return 0;
}
```

INHERITANCE - 7

•THE protected SPECIFIER IS EQUIVALENT TO private AND IS USED TO KEEP A MEMBER private AND AT THE SAME TIME ALLOW A DERIVED CLASS ACCESS TO IT.

•THE DECLARATION FOR protected MEMBERS WITHIN A CLAS IS USUALLY PUT AFTER THE DEFAULT (private) AND BEFORE THE public PARTS. BUT IT COULD BE PUT ANYWHERE WITHIN THE DECLARATION OF THE CLASS.

•protected CAN ALSO BE USED WITH STRUCTURES AND UNIONS.

INHERITANCE - 8

```
class prot1
{    int a;
protected:
      int b;
public:
      int c, d;

// ...
};

main()
{    prot1 o1;

//   o1.b = 3;//an error since protected and
//                  thus private to the class
// ...
     return 0;}
```

INHERITANCE - 9

```
class prot_base
{    int a;
protected:
     int b;
public:
     void init_a(int x) {a = x;}
     void init_b(int y) {b = y;}
};

class derived: public prot_base
{public:
//it has access to b which is protected in base
//but inherited as public thus protected here
//as opposed to a which remains private

     void display_b() {cout << b << endl;}
};
```

INHERITANCE - 10

```
main()
{    derived o1;
//    o1.a = 1;//ERROR, a is private to base
//    o1.b = 2;//ERROR, b is protected to base
//                    and thus private

      o1.init_a(1);
      o1.init_b(2);
      o1.display_b();
//has access to b because it is
//a member function of “derived” and thus
//has access to protected members of
//”prot_base” since it was inherited as
//public

     return 0;
}
```

INHERITANCE - 11

•CONSTRUCTORS ARE EXECUTED IN ORDER OF DERIVATION (BASE CLASS CONSTRUCTOR FIRST, THEN THE DERIVED CLASS CONSTRUCTOR) WHEREAS THE DESTRUCTORS ARE EXECUTED IN REVERSE ORDER.

•YOU CAN PASS ARGUMENTS FROM THE DERIVED CLASS CONSTRUCTOR TO THE BASE CLASS CONSTRUCTOR. THE DERIVED CLASS CAN PASS ALL, SOME, OR NO ARGUMENTS TO THE BASE CLASS DEPENDING ON THE BASE CLASS DECLARATION. HOWEVER, ALL ARGUMENTS MUST BE PASSED TO THE DERIVED CLASS WHICH WILL THEN PASS TO THE BASE CLASS THE ARGUMENTS IT NEEDS.

INHERITANCE - 12

```
#include <iostream.h>

class base
{  int a;
public:
   base(int x){cout << “constructing base\n”;
                    a = x;}
  ~base(){cout << “destructing base\n”;}
};

class derived: public base
{ int b;
public:
  derived(int x, int y) : base(y)//NOTE
  {cout << “constructing derived\n”; b = x;}
  ~derived(){cout<<“destructing derived\n”;}
};
```

INHERITANCE - 13

```
main()
{
    derived o1(3, 8);

    return 0;
}
```

•TEMPLATE FOR PASSING ARGUMENTS TO THE BASE CLASS

***dved_const_name*(*list*):*bse_const_name*(*list*)**
{ // ... *derived constructor body*}

•WHEN A DERIVED CLASS IS USED AS A BASE FOR ANOTHER DERIVED CLASS THE ORIGINAL BASE IS CALLED AND INDIRECT BASE.

INHERITANCE - 14

•**TEMPLATE FOR PASSING ARGUMENTS TO SEVERAL BASE CLASSES**

class ***derived_name*:*access_spec base1_name*,**
***access_spec base2_name*,**
..., *access_spec basen_name*
{ // ... *derived class body* };

***dved_const_name*(*list*):*bse1_const_name*(*list*),**
***bse2_const_name*(*list*),**
...,*bsen_const_name*(*list*)
{ // ... *derived constructor body*}

INHERITANCE - 15

•USE VIRTUAL BASE CLASSES TO PREVENT SEVERAL COPIES OF A BASE FROM BEING PRESENT IN THE DERIVED.

```
class base{//...; public: int i; //...;}

class d1: virtual public base {// ... };

class d2: virtual public base {// ... };

class d3: public d1, public d2{// ... };

main()
{    d3 o1;
     o1.i = 1;//unambiguous since base virtual
     return 0;}
```