**6.1 Function Overloading:-**

 C++ also permits overloading of functions. This means that we can use the same function name to create functions that perform a variety of different tasks. This is known as *function polymorphism* in OOP. We can design a family of functions with one function name but with different argument lists.

 // Declarations

 int add(int a, int b); // prototype 1

 int add(int a, int b, int c); // prototype 2

 double add(double x, double y); // prototype 3

 double add(int p, double q); // prototype 4

 double add(double p, int q); // prototype 5

 // Function calls

 cout << add(5, 10); // uses prototype 1

 cout << add(5, 10.0); // uses prototype 4

 cout << add(12.5, 7.5); // uses prototype 3

 cout << add(5, 10, 15); // uses prototype 2

 cout << add(0.75, 5); // uses prototype 5

A function call first matches the prototype having the same number and type of arguments and then calls the appropriate function for execution. A best match must be unique. The function selection involves the following steps:

1. The compiler first tries to find an exact match in which the types of actual arguments are the same and use that function.
2. If an exact match is not found. The compiler uses the integral promotions to the actual arguments, such as:

 **Char** to **int**

 **Float** to **double**

///////////////////////////////////////[ function overloading ]///////////////////////////////////

Ex : W.P create class to be called **volume** ?

# include <iostream.h>

int volume (int); // prototype declarations

double volume (double, int ); // for overloading volume ( )

long volume (long, int, int );

void vmain ( )

{

 cout <<volume (10) << “\n’’

 cout<<volume (2.5,8)<<’’\n”;

 cout<< volume (100l,75,15);

}

//………………………..FUNCTION DEFINITION……………………….

int volume (int s ) //cube

{

 return (s\*s\*s);

}

double volume (double r, int h ) // cylinder

{

 return (3.14519 \* r\*r\*h);

}

long volume (long l, int b , int h ) // rectangular box

}

return (l\*b\*h)

}

\*\*\*\*\*\*\*\*\* The output of above program would be:

 1000

 157.2595

 112500

 **6.2 Different Numbers of Arguments Example:**

 **The starline() function printed a line using 45 asterisks.**

**The repchar() function used a character and a line length that were both specified when the function was called.**

**The charline() function that always prints 45 characters but that allows the calling program to specify the character to be printed.**

**These three functions—starline(), repchar(), and charline()— perform similar activities but have different names.**

**For programmers using these functions, that means three names to remember and three places to look them up if they are listed alphabetically in an application’s.**

**It would be better to use the same name for all three functions, even though they each have different arguments.**

**Here’s a program, OVERLOAD, that makes this possible:**

**#include <iostream.h>**

**void repchar(); //declarations**

**void repchar(char);**

**void repchar(char, int);**

**int main()**

**{ repchar();**

**repchar(‘=’);**

**repchar(‘+’, 20);**

**return 0;**

**}**

**void repchar()**

**{**

**for(int j=0; j<45; j++)**

**cout << ‘\*’; // always prints asterisk**

**cout << endl;**

**}**

**void repchar(char ch)**

**{**

**for(int j=0; j<45; j++) // always loops 45 times**

**cout << ch; // prints specified character**

**cout << endl;**

**}**

**// repchar()**

**// displays specified number of copies of specified character**

**void repchar(char ch, int n)**

**{for(int j=0; j<n; j++) // loops n times**

**cout << ch; // prints specified character**

**cout << endl;**

**}**

**This program prints out three lines of characters. Here’s the output:**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**===============================**

**++++++++++++++++++++**

**String variables**

 **strings can be variables or constants. Here’s an example that defines a single string variable. It asks the user to enter a string, and places this string in the string variable. Then it displays the string. Here’s the listing for STRINGIN:**

EX: e**Here’s a program, in which three functions all access a global variable.**

**// demonstrates global variables :**

#include <iostream.h>

int main()

{const int MAX = 80; //max characters in string

char str[MAX]; //string variable str

cout << “Enter a string: “;

cin >> str; //put string in str

cout << “You entered: “ << str << endl;

return 0;}

#include <iostream.h>

#include <conio.h> //for getch()

char ch = ‘a’; //global variable ch

void getachar();

void putachar();

int main()

{ while( ch != ‘\r’ ) //main() accesses ch

{ getachar();

putachar();

}

cout << endl;

return 0;

}

void getachar() //getachar() accesses ch

{ ch = getch();}

void putachar() //putachar() accesses ch

{ cout << ch; }

**Ex: A stack**

**#include <iostream.h>**

**class Stack**

**{**

**private:**

**enum { MAX = 10 }; //(non-standard syntax)**

**int st[MAX]; //stack: array of integers**

**int top; //number of top of stack**

**public:**

**Stack() //constructor**

**{ top = 0; }**

**void push(int var) //put number on stack**

**{ st[++top] = var; }**

**int pop() //take number off stack**

**{ return st[top--]; }**

**};**

**int main()**

**{**

**Stack s1;**

**s1.push(11);**

**s1.push(22);**

**cout << “1: “ << s1.pop() << endl; //22**

**cout << “2: “ << s1.pop() << endl; //11**

**s1.push(33);**

**s1.push(44);**

**s1.push(55);**

**s1.push(66);**

**cout << “3: “ << s1.pop() << endl; //66**

**cout << “4: “ << s1.pop() << endl; //55**

**cout << “5: “ << s1.pop() << endl; //44**

**cout << “6: “ << s1.pop() << endl; //33**

**return 0;}**

***comments:***

**The size of the array used for the stack is specified by MAX, in the statement**

**enum { MAX = 10 };** 4:

In keeping with the philosophy of encapsulation, it’s preferable to define constants that will be used entirely within a class, as MAX is here, within the class. Thus the use of global const variables for this purpose is no optimal. Standard C++ mandates that we should be able to declare MAX within the class as static const int MAX = 10;

This means that MAX is constant and applies to all objects in the class.

**Here’s the output:**

**1: 22**

**2: 11**

**3: 66**

**4: 55**

**5: 44**

**6: 33**

**EX: Arrays of English Distances**

**The next program, ENGLARAY, demonstrates an array of such objects.**

#include <iostream.h>

class Distance //English Distance class

{private: int feet; float inches;

public:

void getdist() //get length from user

{cout << “\n Enter feet: “; cin >> feet;

cout << “ Enter inches: “; cin >> inches;}

void showdist() const //display distance

 { cout << feet << “\’-” << inches << endl‘\”’; }

};

 void main()

{ Distance dist[100]; //array of distances

 int n=0; //count the entries

char ans; //user response (‘y’ or ‘n’)

cout << endl;

do { //get distances from user

cout << “Enter distance number “ << n+1;

dist[n++].getdist(); //store distance in array

cout << “Enter another (y/n)?: “;

cin >> ans;

} while( ans != ‘n’ ); //quit if user types ‘n’

for(int j=0; j<n; j++) //display all distances

{cout << “\nDistance number “ << j+1 << “ is “;

dist[j].showdist();}

cout << endl;

return 0; }

***Here’s a sample interaction when the user enters three distances:***

***Enter distance number 1***

***Enter feet: 5***

***Enter inches: 4***

***Enter another (y/n)? y***

***Enter distance number 2***

***Enter feet: 6***

***Enter inches: 2.5***

***Enter another (y/n)? y***

***Enter distance number 3***

***Enter feet: 5***

***Enter inches: 10.75***

***Enter another (y/n)? n***

***Distance number 1 is 5’-4”***

***Distance number 2 is 6’-2.5”***

***Distance number 3 is 5’-10.75”***

Opertator overloading :

 Operator overloading is one of the many exciting features of c++ language.it is an important technique that has enhanced the power of extensibility of c++ .this means that c++ has the ability to provide the operators with a mechanism of giving such special meanings to an operator is known as operator overloading .

OR:

 **Operator overloading is one of the most exciting features of object-oriented programming. It can transform complex, obscure program listings into intuitively obvious ones. can be changed to the much more readable d3 = d1 + d2; The rather forbidding term operator overloading refers to giving the normal C++ operators, such as +, \*, <=, and +=, additional meanings when they are applied to user-defined data types. Normally a = b + c; works only with basic types such as int and float, and attempting to apply it when a, b, and care objects of a user-defined class will cause complaints from the compiler.**











**Overloading Unary Operators**

 Examples of unary operators are the increment and decrement operators ++ and --, and the unary minus, as in -33. In the COUNTER example ―Objects and Classes,‖ we created a class Counter to keep track of a count. Objects of that class were incremented by calling a member function: c1.inc\_count();That did the job, but the listing would have been more readable if we could have used the increment operator ++ instead: ++c1; Let‘s rewrite COUNTER to make this possible. Here‘s the listing for COUNTPP1:

**EX:W.P define a class to increment counter variable with overloading (++ operator )**

**#include <iostream.h>**

**class Counter**

**{**

**private:**

**int count; //count**

**public:**

**Counter() : count(0) { } //constructor**

**int ret\_count() //return count**

**{ return count; }**

**void operator ++ () //increment (prefix)**

**{++count;}**

**};**

**int main()**

**{Counter c1, c2; //define and initialize**

**cout << “\nc1= ” << c1.ret\_count(); //display**

**cout << “\nc2=” << c2.ret\_count();**

**++c1; //increment c1**

**++c2; //increment c2**

**++c2; //increment c2**

**cout << “\nc1=” << c1.ret\_count(); //display again**

**cout << “\nc2=” << c2.ret\_count() << endl;**

**return 0;}**

**In this program we create two objects of class Counter: c1 and c2. The counts in the objects are displayed; they are initially 0. Then, using the overloaded ++ operator, we increment c1 once and c2 twice, and display the resulting values. Here‘s the program‘s output:**

**c1=0 counts are initially 0**

**c2=0**

**c1=1**  **incremented once**

**c2=2**  **incremented twice**

**The statements responsible for these operations are**

**++c1;**

**++c2;**

**++c2;**

**The ++ operator is applied once to c1 and twice to c**



EX: unary minus operator : a minus operator when used as unary ,takes just one operand . it change the sign of an operand when applied to a basic data item .this operator applied



#include <iostream.h>

class space

{ int X,Y,Z;

public:

void getdata (int a ,int b,int c)

{X=a;

 Y=b;

 Z=c;

 }

void display()

{cout <<"X= "<<X<<"\t";

 cout<<"Y "<<Y<<"\t";

 cout <<"Z "<<Z<<"\t";

 }

 void operator -()

 {X= -X; Y=-Y; Z=-Z; }

 };

 void main()

 {

 space S;

 S.getdata (10,-20,30);

 cout<<" S: ";

 S.display();

 -S;

 cout<<"\n S: ";

 S.display ();

 }



Another example:

**Operator Return Values**

**The operator++() function in the COUNTPP1 program has a subtle defect. You will discover it if you use a statement like this in main():**

**c1 = ++c2;**

**The compiler will complain. Why? Because we have defined the ++ operator to have a return type of void in the operator++() function, while in the assignment statement it is being asked to return a variable of type Counter. That is, the compiler is being asked to return whatever value c2 has after being operated on by the ++ operator, and assign this value to c1. So as defined in COUNTPP1, we can‘t use ++ to increment Counter objects in assignments; it must always stand alone with its operand. Of course the normal ++ operator, applied to basic data types such as int, would not have this problem. To make it possible to use our homemade operator++() in**

**assignment expressions, we must provide a way for it to return a value. The next program, COUNTPP2, does just that.**

**// countpp2.cpp**

**// increment counter variable with ++ operator, return value**

**#include <iostream.h>**

**class Counter**

**{**

**private:**

**int count; //count**

**public:**

**Counter() : count(0) //constructor**

**{ }**

**int ret\_count() //return count**

**{ return count; }**

**Counter operator ++ () //increment count**

**{**

**++count; //increment count**

**Counter temp; //make a temporary Counter**

**temp.count = count; //give it same value as this obj**

**return temp; //return the copy**

**}**

**};**

**int main()**

**{**

**Counter c1, c2; //c1=0, c2=0**

**cout << “\nc1=” << c1.ret\_count(); //display**

**cout << “\nc2=” << c2.ret\_count();**

**++c1; //c1=1**

**c2 = ++c1; //c1=2, c2=2**

**cout << “\nc1=” << c1.ret\_count(); //display again**

**cout << “\nc2=” << c2.ret\_count() << endl;**

**return 0;**

**}**

**Here the operator++() function creates a new object of type Counter, called temp, to use as a return value. It increments the count data in its own object as before, then creates the new temp object and assigns count in the new object the same value as in its own object. Finally, it returns the temp object. This has the desired effect. Expressions like**

**++c1**

**now return a value, so they can be used in other expressions, such as**

**c2 = ++c1;**

**as shown in main(), where the value returned from c1++ is assigned to c2. The output from this program is**

**c1=0**

**c2=0**

**c1=2**

**c2=2**

**6.3 Nameless Temporary Objects**

**In COUNTPP2 we created a temporary object of type Counter, named temp, whose sole purpose was to provide a return value for the ++ operator. This required three statements.**

**Counter temp; // make a temporary Counter object**

**temp.count = count; // give it same value as this object**

**return temp; // return it**

**There are more convenient ways to return temporary objects from functions and overloaded operators. Let‘s examine another approach, as shown in the program COUNTPP3:**

**// countpp3.cpp**

**// increment counter variable with ++ operator**

**// uses unnamed temporary object**

**#include <iostream.h>**

**class Counter**

**{**

**private:**

**unsigned int count; //count**

**public:**

**Counter() : count(0) //constructor no args**

**{ }**

**Counter(int c) : count(c) //constructor, one arg**

**{ }**

**int ret\_count() //return count**

**{ return count; }**

**Counter operator ++ () //increment count**

**{**

**++count; // increment count, then return**

**return Counter(count); // an unnamed temporary object**

**} // initialized to this count**

**};**

**int main()**

**{**

**Counter c1, c2; //c1=0, c2=0**

**cout << ―\nc1=‖ << c1.ret\_count(); //display**

**cout << ―\nc2=‖ << c2.ret\_count();**

**++c1; //c1=1**

**c2 = ++c1; //c1=2, c2=2**

**cout << ―\nc1=‖ << c1.ret\_count(); //display again**

**cout << ―\nc2=‖ << c2.ret\_count() << endl;**

**return 0;**

**}**

**In this program a single statement return Counter (count); does what all three statements did in COUNTPP2. This statement**

**creates an object of type Counter. This object has no name; it won‘t be around long enough to need one. This unnamed object is initialized to the value provided by the argument count. But wait: Doesn‘t this require a constructor that takes one argument? It does, and to make this statement work we sneakily inserted just such a constructor into the member function list in COUNTPP3.**

**Counter (int c): count(c) //constructor, one arg**

**{ }**

**Once the unnamed object is initialized to the value of count, it can then be returned. The output of this program is the same as that of COUNTPP2. The approaches in both COUNTPP2 and COUNTPP3 involve making a copy of the original object (the object of which the function is a member), and returning the copy.**

**6.4 Postfix Notation**

 **where the variable is incremented after its value is used in the expression?**

**c1++**

**To make both versions of the increment operator work, we define two overloaded ++ operators, as shown in the POSTFIX program:**

**// postfix.cpp**

**// overloaded ++ operator in both prefix and postfix**

**#include <iostream.h>**

**class Counter**

**{**

**private:**

**unsigned int count; //count**

**public:**

**Counter() : count(0) //constructor no args**

**{ }**

**Counter(int c) : count(c) //constructor, one arg**

**{ }**

**int get\_count() const //return count**

**{ return count; }**

**Counter operator ++ () //increment count (prefix)**

**{ //increment count, then return**

**return Counter(++count); //an unnamed temporary object**

**} //initialized to this count**

**Counter operator ++ (int) //increment count (postfix)**

**{ //return an unnamed temporary**

**return Counter(count++); //object initialized to this**

**} //count, then increment count**

**};**

**int main()**

**{**

**Counter c1, c2; //c1=0, c2=0**

**cout << “\nc1=” << c1.get\_count(); //display**

**cout << “\nc2=” << c2.get\_count();**

**++c1; //c1=1**

**c2 = ++c1; //c1=2, c2=2 (prefix)**

**cout << “\nc1=” << c1.get\_count(); //display**

**cout << “\nc2=” << c2.get\_count();**

**c2 = c1++; //c1=3, c2=2 (postfix)**

**cout << “\nc1=” << c1.get\_count(); //display again**

**cout << “\nc2=” << c2.get\_count() << endl;**

**return 0;**

**}**

**Now there are two different declarators for overloading the ++ operator. The one we‘ve seen before, for prefix notation, is Counter operator ++ () The new one, for postfix notation, is Counter operator ++ (int)**

**The only difference is the int in the parentheses. This int isn‘t really an argument, and it doesn‘t mean integer. It‘s simply a signal to the compiler to create the postfix version of the operator. Here‘s the**

**output from the program:**

**c1=0**

**c2=0**

**c1=2**

**c2=2**

**c1=3**

**c2=2**

**We saw the first four of these output lines in COUNTPP2 and COUNTPP3. But in the last two lines we see the results of the statement**

**c2=c1++;**

**Here, c1 is incremented to 3, but c2 is assigned the value of c1 before it is incremented, so c2 retains the value 2. Of course, you can use this same approach with the decrement operator (--).**

**6.5 Overloading Binary Operators**

**Binary operators can be overloaded just as easily as unary operators. We‘ll look at examples that overload arithmetic operators, comparison operators, and arithmetic assignment operators.**

**6.5.1 Arithmetic Operators**

**In the English Distance program we showed how two English Distance objects could be added using a member function add\_dist():**

**dist3.add\_dist(dist1, dist2);**

**By overloading the + operator we can reduce this dense-looking expression to**

**dist3 = dist1 + dist2;**

**Here‘s the listing for ENGLPLUS, which does just that:**

**// englplus.cpp**

**// overloaded ‗+‘ operator adds two Distances**

**#include <iostream.h>**

**class Distance //English Distance class**

**{**

**private:**

**int feet;**

**float inches;**

**public: //constructor (no args)**

**Distance() : feet(0), inches(0.0)**

**{ } //constructor (two args)**

**Distance(int ft, float in) : feet(ft), inches(in)**

**{ }**

**void getdist() //get length from user**

**{**

**cout << “\nEnter feet: “; cin >> feet;**

**cout << “Enter inches: “; cin >> inches;**

**}**

**void showdist() const //display distance**

**{ cout << feet << “ “ << inches << “ “; }**

**Distance operator + ( Distance ) const; //add 2 distances**

**};**

**//add this distance to d2**

**Distance Distance::operator + (Distance d2) const //return sum**

**{**

**int f = feet + d2.feet; //add the feet**

**float i = inches + d2.inches; //add the inches**

**if(i >= 12.0) //if total exceeds 12.0,**

**{ //then decrease inches**

**i -= 12.0; //by 12.0 and**

**f++; //increase feet by 1**

**} //return a temporary Distance**

**return Distance(f,i); //initialized to sum**

**}**

**int main()**

**{Distance dist1, dist3, dist4; //define distances**

**dist1.getdist(); //get dist1 from user**

**Distance dist2(11, 6.25); //define, initialize dist2**

**dist3 = dist1 + dist2; //single ‗+‘ operator**

**dist4 = dist1 + dist2 + dist3; //multiple ‗+‘ operators**

**//display all lengths**

**cout << “dist1 = “; dist1.showdist(); cout << endl;**

**cout << “dist2 = “; dist2.showdist(); cout << endl;**

**cout << “dist3 = “; dist3.showdist(); cout << endl;**

**cout << “dist4 = “ dist4.showdist(); cout << endl;**

**return 0;**

**}**

**To show that the result of an addition can be used in another addition as well as in an assignment, another addition is performed in main (). We add dist1, dist2, and dist3 to obtain dist4 (which should be double the value of dist3), in the statement**

**dist4 = dist1 + dist2 + dist3;//Nameless Temporary Object will hold the intermediate result from adding dist1 and dist2.**

**Here‘s the output from the program:**

**Enter feet: 10**

**Enter inches: 6.5**

**dist1 = 10‘-6.5 from user**

**dist2 = 11‘-6.25 initialized in program**

**dist3 = 22‘-0.75 dist1+dist2**

**dist4 = 44‘-1.5 dist1+dist2+dist3**

**In class Distance the declaration for the operator+ () function looks like this:**

**Distance operator + (Distance);**

**This function has a return type of Distance, and takes one argument of type Distance. In expressions like**

**dist3 = dist1 + dist2;**

**It‘s important to understand how the return value and arguments of the operator relate to the objects. When the compiler sees this expression it looks at the argument types, and finding only type Distance, it realizes it must use the Distance member function operator+(). But what does this function use as its argument—dist1 or dist2? And doesn‘t it need two arguments, since there are two numbers to be added?**

**Here‘s the key: The argument on the left side of the operator (dist1 in this case) is the object of which the operator is a member. The object on the right side of the operator (dist2) must be furnished as an argument to the operator. The operator returns a value, which canbe assigned or used in other ways; in this case it is assigned to dist3.**

**6.5.2 Overloaded binary operator: one argument*.***

**In the operator+ () function, the left operand is accessed directly—since this is the object of which the operator is a member— using feet and inches. The right operand is accessed as the function‘s argument, as d2.feet and d2.inches. We can generalize and say that an overloaded operator always requires one less argument than its number of operands, since one operand is the object of which the operator is a member. That‘s why unary operators require no arguments. To calculate the return value of operator+() in ENGLPLUS, we first add the feet and inches from the two operands (adjusting for a carry if necessary). The resulting values, f and i, are then used to initialize a nameless Distance object, which is returned in the statement**

**return Distance (f, i);**

**This is similar to the arrangement used in COUNTPP3, except that the constructor takes two arguments instead of one. The statement**

**dist3 = dist1 + dist2;**

**In main () then assigns the value of the nameless Distance object to dist3. Compare this intuitively obvious statement with the use of a function call to perform the same task, as in the ENGLCON example. Similar functions could be created to overload other operators in the Distance class, so you could subtract, multiply, and divide objects of this class in natural-looking ways.**

**6.5 Overloading Binary Operators**

**Binary operators can be overloaded just as easily as unary operators. We‘ll look at examples that overload arithmetic operators, comparison operators, and arithmetic assignment operators.**

**6.5.1 Arithmetic Operators**

**In the English Distance program we showed how two English Distance objects could be added using a member function add\_dist():**

**dist3.add\_dist(dist1, dist2);**

**By overloading the + operator we can reduce this dense-looking expression to**

**dist3 = dist1 + dist2;**

**Here‘s the listing for ENGLPLUS, which does just that:**

**// englplus.cpp**

**// overloaded ‗+‘ operator adds two Distances**

**#include <iostream.h>**

**class Distance //English Distance class**

**{**

**private:**

**int feet;**

**float inches;**

**public: //constructor (no args)**

**Distance() : feet(0), inches(0.0)**

**{ } //constructor (two args)**

**Distance(int ft, float in) : feet(ft), inches(in)**

**{ }**

**void getdist() //get length from user**

**{**

**cout << ―\nEnter feet: ―; cin >> feet;**

**cout << ―Enter inches: ―; cin >> inches;**

**}**

**void showdist() const //display distance**

**{ cout << feet << “\‘-‖” << inches << “\‖‘”; }**

**Distance operator + ( Distance ) const; //add 2 distances**

**};**

**//add this distance to d2**

**Distance Distance::operator + (Distance d2) const //return sum**

**{**

**int f = feet + d2.feet; //add the feet**

**float i = inches + d2.inches; //add the inches**

**if(i >= 12.0) //if total exceeds 12.0,**

**{ //then decrease inches**

**i -= 12.0; //by 12.0 and**

**f++; //increase feet by 1**

**} //return a temporary Distance**

**return Distance(f,i); //initialized to sum**

**}**

**int main()**

**{**

**Distance dist1, dist3, dist4; //define distances**

**dist1.getdist(); //get dist1 from user**

**Distance dist2(11, 6.25); //define, initialize dist2**

**dist3 = dist1 + dist2; //single ‗+‘ operator**

**dist4 = dist1 + dist2 + dist3; //multiple ‗+‘ operators**

**//display all lengths**

**cout << ―dist1 = ―; dist1.showdist(); cout << endl;**

**cout << ―dist2 = ―; dist2.showdist(); cout << endl;**

**cout << ―dist3 = ―; dist3.showdist(); cout << endl;**

**cout << ―dist4 = ―; dist4.showdist(); cout << endl;**

**return 0;**

**}**

**To show that the result of an addition can be used in another addition as well as in an assignment, another addition is performed in main (). We add dist1, dist2, and dist3 to obtain dist4 (which should be double the value of dist3), in the statement**

**dist4 = dist1 + dist2 + dist3;//Nameless Temporary Object will hold the intermediate result from adding dist1 and dist2.**

**Here‘s the output from the program:**

**Enter feet: 10**

**Enter inches: 6.5**

**dist1 = 10‘-6.5‖** **from user**

**dist2 = 11‘-6.25‖** **initialized in program**

**dist3 = 22‘-0.75‖** **dist1+dist2**

**dist4 = 44‘-1.5‖** **dist1+dist2+dist3**

**In class Distance the declaration for the operator+ () function looks like this:**

**Distance operator + (Distance);**

**This function has a return type of Distance, and takes one argument of type Distance. In expressions like**

**dist3 = dist1 + dist2;**

**It‘s important to understand how the return value and arguments of the operator relate to the objects. When the compiler sees this expression it looks at the argument types, and finding only type Distance, it realizes it must use the Distance member function operator+(). But what does this function use as its argument—dist1 or dist2? And doesn‘t it need two arguments, since there are two numbers to be added?**

**Here‘s the key: The argument on the left side of the operator (dist1 in this case) is the object of which the operator is a member. The object on the right side of the operator (dist2) must be furnished as an argument to the operator. The operator returns a value, which can**

**6.5.2 Overloaded binary operator: one argument*.***

**In the operator+ () function, the left operand is accessed directly—since this is the object of which the operator is a member—using feet and inches. The right operand is accessed as the function‘s argument, as d2.feet and d2.inches. We can generalize and say that an overloaded operator always requires one less argument than its number of operands, since one operand is the object of which the operator is a member. That‘s why unary operators require no arguments. To calculate the return value of operator+() in ENGLPLUS, we first add the feet and inches from the two operands (adjusting for a carry if necessary). The resulting values, f and i, are then used to initialize a nameless Distance object, which is returned in the statement**

**return Distance (f, i);**

**This is similar to the arrangement used in COUNTPP3, except that the constructor takes two arguments instead of one. The statement**

**dist3 = dist1 + dist2;**

**In main () then assigns the value of the nameless Distance object to dist3. Compare this intuitively obvious statement with the use of a function call to perform the same task, as in the ENGLCON example. Similar functions could be created to overload other operators in the Distance class, so you could subtract, multiply, and divide objects of this class in natural-looking ways.**

**6.5.3 Arithmetic Assignment Operators**

**Let‘s finish up our exploration of overloaded binary operators with an arithmetic assignment operator: the += operator. Recall that this operator combines assignment and addition into one step. We‘ll use this operator to add one English distance to a second, leaving the result in the first. This is similar to the ENGLPLUS example shown earlier, but there is a subtle difference.**

**#include <iostream.h>**

**class Distance //English Distance class**

**{private:**

**int feet;**

**float inches;**

**public:**

**Distance() : feet(0), inches(0.0)**

**{ }**

**Distance(int ft, float in) : feet(ft), inches(in)**

**{ }**

**void getdist()**

**{**

**cout << “\nEnter feet: “; cin >> feet;**

**cout << “Enter inches: “; cin >> inches;**

**}**

**void showdist() const //display distance**

**{ cout << feet << “ ” << inches << “ “; }**

**void operator += ( Distance );**

**};**

**void Distance::operator += (Distance d2)**

**{feet += d2.feet;**

**inches += d2.inches;**

**if(inches >= 12.0)**

**{**

**inches -= 12.0;**

**feet++;**

**}**

**}**

**void main()**

**{Distance dist1;**

**dist1.getdist();**

**cout << “\ndist1 = “; dist1.showdist();**

**Distance dist2(11, 6.25);**

**cout << “\ndist2 = “; dist2.showdist();**

**dist1 += dist2;**

**cout << “\nAfter addition,”;**

**cout << “\ndist1 = “; dist1.showdist();**

**cout << endl;**

**;}**

**In this program we obtain a distance from the user and add to it a second distance, initialized to 11'–6.25'' by the program. Here‘s a sample of interaction with the program:**

**Enter feet: 3**

**Enter inches: 5.75**

**dist1 = 3‘-5.75‖**

**dist2 = 11‘-6.25‖**

**After addition,**

**dist1 = 15**

**In this program the addition is carried out in main() with the statement**

**dist1 += dist2;**

**This causes the sum of dist1 and dist2 to be placed in dist1. Notice the difference between the function used here, operator+=(), and that used in ENGLPLUS, operator+(). In the earlier operator+() function, a new object of type Distance had to be created and returned by the function so it could be assigned to a third Distance object, as in**

**dist3 = dist1 + dist2;**

**In the operator+=() function in ENGLPLEQ, the object that takes on the value of the sum is the object of which the function is a member. Thus it is feet and inches that are given values, not temporary variables used only to return an object. The operator+= () function has no return value; it returns type void. A return value is not necessary with arithmetic assignment operators such as +=, because the result of the assignment operator is not assigned to anything. The operator is used alone, in expressions like the one in the program.**

**dist1 += dist2;**

**Exercises**

1. **To the Distance class in the ENGLPLUS program in this chapter, add an overloaded - operator that subtracts two distances. It should allow statements like dist3= dist1-dist2;. Assume that the operator will never be used to subtract a larger number from a smaller one (that is, negative distances are not allowed).**

**2. Modify the time class from Exercise 3 in Chapter 6 so that instead of a function add\_time ( ) it uses the overloaded + operator to add two times. Write a program to test this class.**

**3. Create a class Int Overload four integer arithmetic operators (+, -, \*, and /) so that they operate on objects of type Int. If the result of any such arithmetic operation exceeds the normal range of ints (in a 32-bit environment)— from 2,147,483,648 to –2,147,483,647—have the operator print a warning and terminate the program. Such a data type might be useful where mistakes caused by arithmetic overflow are unacceptable. Hint: To facilitate checking for overflow, perform the calculations using type long double. Write a program to test this class.**

**4. Augment the time class referred to in Exercise 3 to include overloaded increment (++) and decrement (--) operators that operate in both prefix and postfix notation and return values. Add statements to main() to test these operators.**

**5. Add to the time class of Exercise 5 the ability to subtract two time values using the overloaded (-) operator, and to multiply a time value by a number of type float, using the overloaded (\*) operator.**

**Answers to Exercises**

**1).**

**// ex8\_1.cpp**

**// overloaded ‗-‘ operator subtracts two Distances**

**#include <iostream.h>**

**class Distance //English Distance class**

**{**

**private:**

**int feet;**

**float inches;**

**public: //constructor (no args)**

**Distance() : feet(0), inches(0.0)**

**{ } //constructor (two args)**

**Distance(int ft, float in) : feet(ft), inches(in) { }**

**void getdist() //get length from user**

**{**

**cout << ―\nEnter feet: ―; cin >> feet;**

**cout << ―Enter inches: ―; cin >> inches;**

**}**

**void showdist() //display distance**

**{ cout << feet << ―\‘-‖ << inches << ‗\‖‘; }**

**Distance operator + ( Distance ); //add two distances**

**Distance operator - ( Distance ); //subtract two distances**

**};**

**//add d2 to this distance**

**Distance Distance::operator + (Distance d2) //return the sum**

**{**

1. **int f = feet + d2.feet; //add the feet**

**float i = inches + d2.inches; //add the inches**

**if(i >= 12.0) //if total exceeds 12.0,**

**{ //then decrease inches**

**i -= 12.0; //by 12.0 and**

**f++; //increase feet by 1**

**} //return a temporary Distance**

**return Distance(f,i); //initialized to sum**

**}**

**//subtract d2 from this dist**

**Distance Distance::operator - (Distance d2) //return the diff**

**{**

**int f = feet - d2.feet; //subtract the feet**

**float i = inches - d2.inches; //subtract the inches**

**if(i < 0) //if inches less than 0,**

**{ //then increase inches**

**i += 12.0; //by 12.0 and**

**f--; //decrease feet by 1**

**} //return a temporary Distance**

**return Distance(f,i); //initialized to difference**

**}**

**int main()**

**{**

**Distance dist1, dist3; //define distances**

**dist1.getdist(); //get dist1 from user**

**Distance dist2(3, 6.25); //define, initialize dist2**

**dist3 = dist1 - dist2; //subtract**

**//display all lengths**

**cout << ―\ndist1 = ―; dist1.showdist();**

**cout << ―\ndist2 = ―; dist2.showdist();**

**cout << ―\ndist3 = ―; dist3.showdist();**

**cout << endl;**

**return 0; }**

**2).**

**// ex8\_3.cpp**

**// overloaded ‗+‘ operator adds two times**

**#include <iostream.h>**

**class time**

**{**

**private:**

**int hrs, mins, secs;**

**public:**

**time() : hrs(0), mins(0), secs(0) //no-arg constructor**

**{ } //3-arg constructor**

**time(int h, int m, int s) : hrs(h), mins(m), secs(s)**

**{ }**

**void display() //format 11:59:59**

**{ cout << hrs << ―:‖ << mins << ―:‖ << secs; }**

**time operator + (time t2) //add two times**

**{int s = secs + t2.secs; //add seconds**

1. **int m = mins + t2.mins; //add minutes**

**int h = hrs + t2.hrs; //add hours**

**if( s > 59 ) //if secs overflow,**

**{ s -= 60; m++; } // carry a minute**

**if( m > 59 ) //if mins overflow,**

**{ m -= 60; h++; } // carry an hour**

**return time(h, m, s); //return temp value**

**}**

**};**

**int main()**

**{time time1(5, 59, 59); //create and initialize**

**time time2(4, 30, 30); // two times**

**time time3; //create another time**

**time3 = time1 + time2; //add two times**

**cout << “\ntime3 = “; time3.display();**

**cout << endl;**

**return 0;**

**}**





#include <iostream.h>

class complex

{

 float X, Y;

 public:

 complex ()

 { }

 complex (float a ,float b)

 { X=a ; Y=b;}

 complex operator + (complex C)

 { complex temp;

 temp.X =X+ C.X;

 temp.Y= Y+ C.Y;

 return (temp);

 }

void display ()

{ cout << X <<" + "<< Y<<endl; }

};

void main ()

{

 complex C1 ,C2 ,C3;

 C1=complex (2.5,3.5);

 C2=complex (1.6, 2.7);

 C3=C1 + C2;

 cout<< "C1= "; C1.display(); cout<< "C2= "; C2.display(); cout<< "C3= "; C3.display(); }

















**Function Overloading**

* **C++ supports writing more than one function with the same name but different argument lists. This could include:**
	+ **different data types**
	+ **different number of arguments**
* **The advantage is that the same apparent function can be called to perform similar but different tasks. The following will show an example of this.**

**void** swap **(int** \*a, **int** \*b**)** **;**

**void** swap **(float** \*c, **float** \*d**)** **;**

**void** swap **(char** \*p, **char** \*q**)** **;**

**int main ( )**

**{**

 **int** a = 4, b = 6 **;**

 **float** c = 16.7, d = -7.89 **;**

 **char** p = 'M' , q = 'n' **;**

 swap **(**&a, &b**) ;**

 swap **(**&c, &d**)** **;**

 swap **(**&p, &q**)** **;**

**void** swap **(int** \*a, **int** \*b**)**

**{ int** temp**;** temp = \*a**;** \*a = \*b**;** \*b = temp**; }**

**void** swap **(float** \*c, **float** \*d**)**

**{** **float** temp**;** temp = \*c**;** \*c = \*d**;** \*d = temp**;** **}**

**void** swap **(char** \*p, **char** \*q**)**

**{** **char** temp**;** temp = \*p**;**  \*p = \*q**;** \*q = temp**;** **}**

**class Complex {**

**public:**

 **// Constructor**

 **Complex (double r, double i);**

**friend Complex operator\***

**(const Complex &, const Complex &);**

**// but not friend Complex operator^**

**// (const Complex &, const Complex &);**

**private:**

 **int real\_;**

 **int imaginary\_;**

**};**

**// multiplication works just fine**

**Complex operator\* (const Complex &,**

 **const Complex &);**

**// exponentiation operator unworkable**

**// Complex operator^ (const Complex &,**

**// const Complex &);**

**// declarations in .h file**

**class A {**

**public:**

 **friend bool operator<**

 **(const A &, const A &);**

 **A operator++(); // prefix**

 **A operator++(int); // postfix**

**private:**

 **int m\_;**

**};**

**bool operator==(const A &lhs,**

 **const A &rhs);**

**// definitions in .cpp file**

**bool operator==(const A &lhs,**

 **const A &rhs)**

**{return lhs.m\_ == rhs.m\_;}**

**A A::operator++() // prefix**

**{++m\_; return \*this;}**

**A A::operator++(int) // postfix**

**{A ret(\*this); ++\*this; return ret;}**

Complex C1 (3, 5), C2 (5, 9), C3;

 C3 = C1 + C2; // addition

 C2 = C3 \* C1; // subtraction

 C1 = -C2; // negation

* For user-defined types, when you use an operator, you are making a function call.
* Consider the expression: C2 + C1
	+ This is translated into a function call.
	+ The name of the function is “operator+”
	+ The call is:

C2.operator+(C1);

Declaring operator+
As a Member Function

class Complex {

 public:

 const Complex

 operator+ (const Complex &rhs) const;

 …

};

* Note all of the const’s!
* const Complex
* Complex :: operator+ (const Complex &rhs) const
* {
* Complex sum;
* // accessor and mutators not required
* sum.imagine = imagine + rhs.imagine;
* // but preferred
* sum.setReal( getReal( ) + rhs.getReal ( ) );
* return sum;
* We can now write

 C3 = C2 + C1;

* We can also use **cascading operators**.

 C4 = C3 + C2 + C1;

* And we can write

 C3 = C2 + 7;

* But C3 = 7 + C2 is a compiler error. (Why?)

Overloading Unary Operators

Complex C1(4, 5), C2;

 C2 = -C1;

is an example of a unary operator (minus).

* We can and should overload this operator as a member function.
* C++ operators that can be overloaded

* C++ Operators that cannot be overloaded

Restrictions on Operator Overloading

* Overloading restrictions
	+ Precedence of an operator cannot be changed
	+ Associativity of an operator cannot be changed
	+ Arity (number of operands) cannot be changed
		- Unary operators remain unary, and binary operators remain binary
		- Operators **&**, **\***, **+** and **-** each have unary and binary versions
		- Unary and binary versions can be overloaded separately
* No new operators can be created
	+ Use only existing operators
* No overloading operators for built-in types
	+ Cannot change how two integers are added
	+ Produces a syntax error

Overloading ++ and --

* Pre/post incrementing/decrementing operators
	+ Allowed to be overloaded
	+ Distinguishing between pre and post operators
		- prefix versions are overloaded the same as other prefix unary operators

**d1.operator++(); // for ++d1**

* + - convention adopted that when compiler sees postincrementing expression, it will generate the member-function call

**d1.operator++( 0 ); // for d1++**

* + - **0** is a dummy value to make the argument list of **operator++** distinguishable from the argument list for **++operator**