Object Oriented Programming
14CS55

UNIT - 2

Function Overloading : Function overloading is the process of using the same name for two or more functions. The secret to overloading is that each redefinition of the function must use either different types of parameters or a different number of parameters. It is only through these differences that the compiler knows which function to call in any given situation.

This program overloads myfunc( ) by using different types of parameters

```
#include <iostream>
using namespace std;
int myfunc(int i); // these differ in types of parameters
double myfunc(double i);
int main()
{
cout << myfunc(10) << " "; // calls myfunc(int i)
cout << myfunc(5.4); // calls myfunc(double i)
return 0;
}
double myfunc(double i)
{
return i;
}
int myfunc(int i)
{
return i;
}
```

This program overloads myfunc( ) using a different number of parameters:

```
#include <iostream>
using namespace std;
int myfunc(int i); // these differ in number of parameters
int myfunc(int i, int j);
int main()
{
cout << myfunc(10) << " "; // calls myfunc(int i)
cout << myfunc(4, 5); // calls myfunc(int i, int j)
return 0;
}
int myfunc(int i)
{
```

```
return i;
}
int myfunc(int i, int j)
{
return i*j;
}
```

As mentioned, the key point about function overloading is that the functions must differ in regard to the types and/or number of parameters. Two functions differing only in their return types cannot be overloaded

Overloading Constructor Functions

Constructor functions can be overloaded; in fact, overloaded constructors are very common
There are three main reasons why you will want to overload a constructor function:

- to gain flexibility,
- to allow both initialized and uninitialized objects to be created, and
- to define copy constructors

Overloading a Constructor to Gain Flexibility

Many times you will create a class for which there are two or more possible ways to construct an object. In these cases, you will want to provide an overloaded constructor function for each way

By providing a constructor for each way that a user of your class may plausibly want to construct an object, you increase the flexibility of your class.

Consider this program that creates a class called date, which holds a calendar date. Notice that the constructor is overloaded two ways:

```
#include <iostream>
#include <cstdio>
using namespace std;
class date {
int day, month, year;
public:
date(char *d);
date(int m, int d, int y);
void show_date();
};
// Initialize using string.
date::date(char *d)
{
sscanf(d, "%d%*c%d%*c%d", &month, &day, &year);
```

```
}
// Initialize using integers.
date::date(int m, int d, int y)
{
day = d;
month = m;
year = y;
}
void date::show_date()
{
cout << month << "/" << day;
cout << "/" << year << "\n";
}
int main()
{
date ob1(12, 4, 2001), ob2("10/22/2001");
ob1.show_date();
ob2.show_date();
return 0;
}
```

Allowing Both Initialized and Uninitialized Objects

Another common reason constructor functions are overloaded is to allow both initialized and uninitialized objects (or, more precisely, default initialized objects) to be created

For example, the following program declares two arrays of type powers; one is initialized and the other is not. It also dynamically allocates an array.

```
#include <iostream>
#include <new>
using namespace std;
class powers {
int x;
public:
// overload constructor two ways
powers() { x = 0; } // no initializer
powers(int n) { x = n; } // initializer
int getx() { return x; }
void setx(int i) { x = i; }
};
int main()
{
powers ofTwo[] = {1, 2, 4, 8, 16}; // initialized
powers ofThree[5]; // uninitialized
```

```
powers *p;
int i;
// show powers of two
cout << "Powers of two: ";
for(i=0; i<5; i++) {
cout << ofTwo[i].getx() << " ";
}
cout << "\n\n";
// set powers of three
ofThree[0].setx(1);
ofThree[1].setx(3);
ofThree[2].setx(9);
ofThree[3].setx(27);
ofThree[4].setx(81);
// show powers of three
cout << "Powers of three: ";
for(i=0; i<5; i++) {
cout << ofThree[i].getx() << " ";
}
cout << "\n\n";
// dynamically allocate an array
try {
p = new powers[5]; // no initialization
} catch (bad_alloc xa) {
cout << "Allocation Failure\n";
return 1;
}
// initialize dynamic array with powers of two
for(i=0; i<5; i++) {
p[i].setx(ofTwo[i].getx());
}
// show powers of two
cout << "Powers of two: ";
for(i=0; i<5; i++) {
cout << p[i].getx() << " ";
}
cout << "\n\n";
delete [] p;
return 0;
}
```

In this example, both constructors are necessary. The default constructor is used to construct the uninitialized ofThree array and the dynamically allocated array. The parameterized constructor is called to create the objects for the ofTwo array.

Copy Constructors

One of the more important forms of an overloaded constructor is the copy constructor.

When is copy constructor called?

In C++, a Copy Constructor may be called in following cases:

1. When an object of the class is returned by value.
2. When an object of the class is passed (to a function) by value as an argument.
3. When an object is constructed based on another object of the same class.
4. When compiler generates a temporary object.

It is however, not guaranteed that a copy constructor will be called in all these cases, because the C++ Standard allows the compiler to optimize the copy away in certain cases,

When is user defined copy constructor needed?

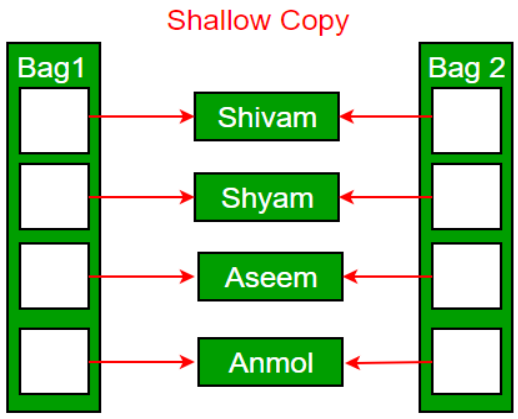
If we don't define our own copy constructor, the C++ compiler creates a default copy constructor for each class which does a member wise copy between objects. The compiler created copy constructor works fine in general. We need to define our own copy constructor only if an object has pointers or any run time allocation of resource like file handle, a network connection..etc.

When a copy constructor exists, the default, bitwise copy is bypassed. The most common general form of a copy constructor is

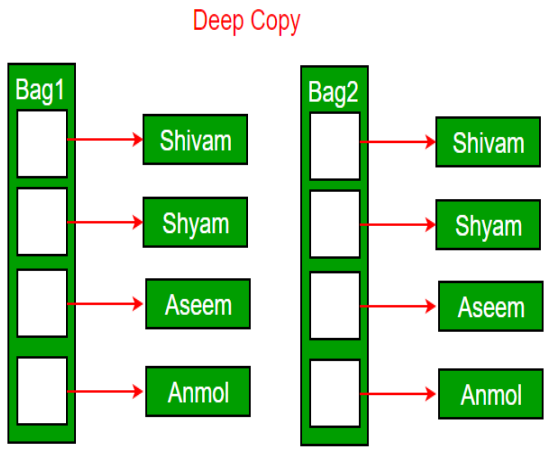
```
classname (const classname &o) {
// body of constructor
}
```

Here, o is a reference to the object on the right side of the initialization

Default constructor does only ***shallow copy.***



Deep copy is possible only with user defined copy constructor.In user defined copy constructor, we make sure that pointers (or references) of copied object point to new memory locations.



Copy constructor vs Assignment Operator

Which of the following two statements call copy constructor and which one calls assignment operator?

```
MyClass t1, t2;
MyClass t3 = t1;  // ----> (1)
t2 = t1;          // -----> (2)
```

Copy constructor is called when a new object is created from an existing object, as a copy of the existing object. Assignment operator is called when an already initialized object is assigned a new value from another existing object. In the above example (1) calls copy constrictor and (2) calls assignment operator.

Write an example class where copy constructor is needed?

Following is a complete C++ program to demonstrate use of Copy constructor. In the following String class, we must write copy constructor.

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

class String
{
private:
        char *s;
        int size;
public:
        String(const char *str = NULL); // constructor
        ~String() { delete [] s; }// destructor
        String(const String&); // copy constructor
        void print() { cout << s << endl; } // Function to print string
        void change(const char *);  // Function to change
};

String::String(const char *str)
{
        size = strlen(str);
        s = new char[size+1];
        strcpy(s, str);
}

void String::change(const char *str)
{
        delete [] s;
        size = strlen(str);
        s = new char[size+1];
        strcpy(s, str);
}

String::String(const String& old_str)
{
        size = old_str.size;
        s = new char[size+1];
        strcpy(s, old_str.s);
}

int main()
{
        String str1("GeeksQuiz");
```

```
        String str2 = str1;

        str1.print(); // what is printed ?
        str2.print();

        str2.change("GeeksforGeeks");

        str1.print(); // what is printed now ?
        str2.print();
        return 0;
}
```

Example 2 :

/* This program creates a "safe" array class. Since space for the array is allocated using new, a copy constructor is provided to allocate memory when one array object is used to initialize another. */

```
#include <iostream>
#include <new>
#include <cstdlib>
using namespace std;
class array {
int *p;
int size;
public:
array(int sz) {
try {
p = new int[sz];
} catch (bad_alloc xa) {
cout << "Allocation Failure\n";
exit(EXIT_FAILURE);
}
size = sz;
}
~array() { delete [] p; }
// copy constructor
array(const array &a);
void put(int i, int j) {
if(i>=0 && i<size) p[i] = j;
}
int get(int i) {
return p[i];
}
};
// Copy Constructor
```

```
array::array(const array &a) {
int i;
try {
p = new int[a.size];
} catch (bad_alloc xa) {
cout << "Allocation Failure\n";
exit(EXIT_FAILURE);
}
for(i=0; i<a.size; i++) p[i] = a.p[i];
}
int main()
{
array num(10);
int i;
for(i=0; i<10; i++) num.put(i, i);
for(i=9; i>=0; i--) cout << num.get(i);
cout << "\n";
// create another array and initialize with num
array x(num); // invokes copy constructor
for(i=0; i<10; i++) cout << x.get(i);
return 0;
}
```

Remember that the copy constructor is called only for initializations. For example, this sequence does not call the copy constructor defined in the preceding program:

```
array a(10); // ...
array b(10);
b = a;  // does not call copy constructor
```

Default Function Arguments

C++ allows a function to assign a parameter a default value when no argument corresponding to that parameter is specified in a call to that function. The default value is specified in a manner syntactically similar to a variable initialization

Example :

```
void myfunc(double d = 0.0)
{
   // ...
}
```

Now, myfunc( ) can be called one of two ways, as the following examples show:

```
myfunc(198.234); // pass an explicit value
```

myfunc(); // let function use default

Example :

```
#include <iostream>
using namespace std;
void clrscr(int size=25);
int main()
{
register int i;

for(i=0; i<30; i++ ) cout << i << endl;
cin.get();
clrscr(); // clears 25 lines
for(i=0; i<30; i++ ) cout << i << endl;
cin.get();
clrscr(10); // clears 10 lines
return 0;
}
void clrscr(int size)
{
for(; size; size--) cout << endl;
}
```

Operator Overloading

In C++, you can overload most operators so that they perform special operations relative to classes that you create. For example, a class that maintains a stack might overload + to perform a push operation and – – to perform a pop. When an operator is overloaded, none of its original meanings are lost. Instead, the type of objects it can be applied to is expanded

Creating a Member Operator Function :

A member operator function takes this general form:

```
ret-type class-name::operator#(arg-list)
{
// operations
}
```

- Often, operator functions return an object of the class they operate on, but ret-type can be any valid type
- The # is a placeholder
- When you create an operator function, substitute the operator for the #

Example : Overloading + operator

```
#include <iostream>
using namespace std;
class loc {
int longitude, latitude;
public:
loc() {}
loc(int lg, int lt) {
longitude = lg;
latitude = lt;
}
void show() {
cout << longitude << " ";
cout << latitude << "\n";
}
loc operator+(loc op2);
};
// Overload + for loc.
loc loc::operator+(loc op2)
{
loc temp;
temp.longitude = op2.longitude + longitude;
temp.latitude = op2.latitude + latitude;
return temp;
}
int main()
{
loc ob1(10, 20), ob2( 5, 30);
ob1.show(); // displays 10 20
ob2.show(); // displays 5 30
ob1 = ob1 + ob2;
ob1.show(); // displays 15 50
return 0;
}
```

- As you can see, operator+( ) has only one parameter even though it overloads the binary + operator
- The reason that operator+( ) takes only one parameter is that the operand on the left side of the + is passed implicitly to the function through the **this** pointer

- The operand on the right is passed in the parameter op2
- When binary operators are overloaded, it is the object on the left that generates the call to the operator function

Further, having operator+( ) return an object of type loc makes possible the following statement:

(ob1+ob2).show(); // displays outcome of ob1+ob2

In this situation, ob1+ob2 generates a temporary object that ceases to exist after the call to show( ) terminates.

Example : Overoading of the+, the –, the =, and the unary ++.

```
#include <iostream>
using namespace std;
class loc {
int longitude, latitude;
public:
loc() {} // needed to construct temporaries
loc(int lg, int lt) {
longitude = lg;
latitude = lt;
}
void show() {
cout << longitude << " ";
cout << latitude << "\n";
}


loc operator+(loc op2);
loc operator-(loc op2);
loc operator=(loc op2);
loc operator++();
};
// Overload + for loc.
loc loc::operator+(loc op2)
{
loc temp;
temp.longitude = op2.longitude + longitude;
temp.latitude = op2.latitude + latitude;
return temp;
}
// Overload - for loc.
loc loc::operator-(loc op2)
{
```

```
loc temp;
// notice order of operands
temp.longitude = longitude - op2.longitude;
temp.latitude = latitude - op2.latitude;
return temp;
}
// Overload asignment for loc.
loc loc::operator=(loc op2)
{
longitude = op2.longitude;
latitude = op2.latitude;
return *this; // i.e., return object that generated call
}
// Overload prefix ++ for loc.
loc loc::operator++()
{
longitude++;
latitude++;
return *this;
}

int main()
{
loc ob1(10, 20), ob2( 5, 30), ob3(90, 90);
ob1.show();
ob2.show();
++ob1;
ob1.show(); // displays 11 21
ob2 = ++ob1;
ob1.show(); // displays 12 22
ob2.show(); // displays 12 22
ob1 = ob2 = ob3; // multiple assignment
ob1.show(); // displays 90 90
ob2.show(); // displays 90 90
return 0;
}
```

Notice that the operator=( ) function returns *this, which is the object that generated the call. This arrangement is necessary if you want to be able to use multiple assignment operations such as this:
ob1 = ob2 = ob3; // multiple assignment

Creating Prefix and Postfix Forms of the Increment and Decrement Operators

```
loc operator++(int x);
```

If the ++ precedes its operand, the operator++( ) function is called. If the ++ follows its operand, the operator++(int x) is called and x has the value zero.

```
// Prefix increment
type operator++( ) {
// body of prefix operator
}

// Postfix increment
type operator++(int x) {
// body of postfix operator
}
// Prefix decrement
type operator– –( ) {
// body of prefix operator
}
// Postfix decrement
type operator– –(int x) {
// body of postfix operator
}
```

Overloading the Shorthand Operators

You can overload any of C++'s "shorthand" operators, such as +=, –=, and the like. For example, this function overloads += relative to loc:

```
loc loc::operator+=(loc op2)
{
longitude = op2.longitude + longitude;
latitude = op2.latitude + latitude;
return *this;
}
```

Operator Overloading Restrictions

- You cannot change the number of operands that an operator takes. (You can choose to ignore an operand, however.)
- operator (described later), operator functions cannot have default arguments.
- these operators cannot be overloaded: . : : .* ?

Operator Overloading Using a Friend Function

You can overload an operator for a class by using a nonmember function, which is usually a friend of the class

Since a friend function is not a member of the class, it does not have a this pointer. Therefore, an overloaded friend operator function is passed the operands explicitly

In this program, the operator+( ) function is made into a friend:

```
#include <iostream>
using namespace std;
class loc {
int longitude, latitude;
public:
loc() {} // needed to construct temporaries
loc(int lg, int lt) {
longitude = lg;
latitude = lt;
}
void show() {
cout << longitude << " ";
cout << latitude << "\n";
}
friend loc operator+(loc op1, loc op2); // now a friend
loc operator-(loc op2);
loc operator=(loc op2);
loc operator++();
};
// Now, + is overloaded using friend function.
loc operator+(loc op1, loc op2)
{
loc temp;
temp.longitude = op1.longitude + op2.longitude;
temp.latitude = op1.latitude + op2.latitude;
return temp;
}
// Overload - for loc.
loc loc::operator-(loc op2)
{
loc temp;
// notice order of operands
temp.longitude = longitude - op2.longitude;
temp.latitude = latitude - op2.latitude;
return temp;
}
// Overload assignment for loc.
loc loc::operator=(loc op2)
```

```
{
longitude = op2.longitude;
latitude = op2.latitude;
return *this; // i.e., return object that generated call
}
// Overload ++ for loc.
loc loc::operator++()
{
longitude++;
latitude++;
return *this;
}
int main()
{
loc ob1(10, 20), ob2( 5, 30);
ob1 = ob1 + ob2;
ob1.show();
return 0;
}
```

There are some restrictions that apply to friend operator functions.

- First, you may not overload the =, ( ), [ ], or –> operators by using a friend function.
- Second, when overloading the increment or decrement operators, you will need to use a reference parameter when using a friend function.

Using a Friend to Overload ++ or – –

friend operator function has no way to modify the operand. Since the friend operator function is not passed a this pointer to the operand, but rather a copy of the operand, no changes made to that parameter affect the operand that generated the call

```
#include <iostream>
using namespace std;
class loc {
int longitude, latitude;
public:
loc() {}
loc(int lg, int lt) {
longitude = lg;
latitude = lt;
}
void show() {
cout << longitude << " ";
cout << latitude << "\n";
```

```
}
loc operator=(loc op2);
friend loc operator++(loc &op);
friend loc operator--(loc &op);
};
// Overload assignment for loc.
loc loc::operator=(loc op2)
{
longitude = op2.longitude;
latitude = op2.latitude;
return *this; // i.e., return object that generated call
}
// Now a friend; use a reference parameter.
loc operator++(loc &op)
{
op.longitude++;
op.latitude++;
return op;
}
// Make op-- a friend; use reference.
loc operator--(loc &op)
{
op.longitude--;
op.latitude--;
return op;
}
int main()
{
loc ob1(10, 20), ob2;
ob1.show();
++ob1;
ob1.show(); // displays 11 21
ob2 = ++ob1;
ob2.show(); // displays 12 22
--ob2;
ob2.show(); // displays 11 21
return 0;
}
```

If you want to overload the postfix versions of the increment and decrement operators using a friend, simply specify a second, dummy integer parameter. For example, this shows the prototype for the friend, postfix version of the increment operator relative to loc.

```
// friend, postfix version of ++
friend loc operator++(loc &op, int x);
```

Friend Operator Functions Add Flexibility

Overloading new and delete

It is possible to overload new and delete. You might choose to do this if you want to use some special allocation method

You might choose to do this if you wan to use some special allocation method. For example, you may want allocation routines that automatically begin using a disk file as virtual memory when the heap has been exhausted. Whatever the reason, it is a very simple matter to overload these operators.

```
// Allocate an object.
void *operator new(size_t size)
{
/* Perform allocation. Throw bad_alloc on failure.
Constructor called automatically. */
return pointer_to_memory;
}
// Delete an object.
void operator delete(void *p)
{
/* Free memory pointed to by p.
Destructor called automatically. */
}
```

- The type size_t is a defined type capable of containing the largest single piece of memory that can be allocated. (size_t is essentially an unsigned integer.)
- The parameter size will contain the number of bytes needed to hold the object being allocated
- The
- overloaded new function must return a pointer to the memory that it allocates, or sthrow a bad_alloc exception if an allocation error occurs

- The delete function receives a pointer to the region of memory to be freed. It then releases the previously allocated memory back to the system. When an object is deleted, its destructor function is automatically called.

Example

```
#include <iostream>
#include <cstdlib>
#include <new>
using namespace std;
class loc {
int longitude, latitude;
public:
loc() {}
loc(int lg, int lt) {
longitude = lg;
latitude = lt;
}
void show() {
cout << longitude << " ";
cout << latitude << "\n";
}
void *operator new(size_t size);
void operator delete(void *p);
};
// new overloaded relative to loc.
void *loc::operator new(size_t size)
{
void *p;
cout << "In overloaded new.\n";
p = malloc(size);
if(!p) {
bad_alloc ba;
throw ba;
}
return p;
}
// delete overloaded relative to loc.
void loc::operator delete(void *p)
{
cout << "In overloaded delete.\n";
free(p);
}
int main()
{
```

```
loc *p1, *p2;
try {
p1 = new loc (10, 20);
} catch (bad_alloc xa) {
cout << "Allocation error for p1.\n";
return 1;
}
try {
p2 = new loc (-10, -20);
} catch (bad_alloc xa) {
cout << "Allocation error for p2.\n";
return 1;;
}
p1->show();
p2->show();
delete p1;
delete p2;
return 0;
}
```

Output from this program is shown here.

```
In overloaded new
In overloaded new
10 20
-10 -20
In overloaded delete
In overloaded delete
```

When new and delete are for a specific class, the use of these operators on any other type of data causes the original new or delete to be employed.

This means that if you add this line to the main( ), the default new will be executed:

int *f = new float; // uses default new

You can overload new and delete globally by overloading these operators outside of any class declaration. When new and delete are overloaded globally, C++'s default new and delete are ignored and the new operators are used for all allocation requests

To see an example of overloading new and delete globally, examine this program:

```
#include <iostream>
#include <cstdlib>
#include <new>
using namespace std;
class loc {
int longitude, latitude;
```

```
public:
loc() {}
loc(int lg, int lt) {
longitude = lg;
latitude = lt;
}
void show() {
cout << longitude << " ";
cout << latitude << "\n";
}
};
// Global new
void *operator new(size_t size)
{
void *p;
p = malloc(size);
if(!p) {
bad_alloc ba;
throw ba;
}
return p;
}
// Global delete
void operator delete(void *p)
{
free(p);
}
int main()
{
loc *p1, *p2;
float *f;
try {
p1 = new loc (10, 20);
} catch (bad_alloc xa) {
cout << "Allocation error for p1.\n";
return 1;;
}
try {
p2 = new loc (-10, -20);
} catch (bad_alloc xa) {
cout << "Allocation error for p2.\n";
return 1;;
}
try {
f = new float; // uses overloaded new, too
} catch (bad_alloc xa) {
```

```
cout << "Allocation error for f.\n";
return 1;;
}
*f = 10.10F;
cout << *f << "\n";
p1->show();
p2->show();
delete p1;
delete p2;
delete f;
return 0;
}
```

Overloading new and delete for Arrays

If you want to be able to allocate arrays of objects using your own allocation system, you will need to overload new and delete a second time

```
// Allocate an array of objects.
void *operator new[](size_t size)
{
/* Perform allocation. Throw bad_alloc on failure.
Constructor for each element called automatically. */
return pointer_to_memory;
}
// Delete an array of objects.
void operator delete[](void *p)
{
/* Free memory pointed to by p.
Destructor for each element called automatically.
*/
}
```

When allocating an array, the constructor function for each object in the array is automatically called. When freeing an array, each object's destructor is automatically called. You do not have to provide explicit code to accomplish these actions.

```
#include <iostream>
#include <cstdlib>
#include <new>
using namespace std;
class loc {
int longitude, latitude;
public:
```

```
loc() {longitude = latitude = 0;}
loc(int lg, int lt) {
longitude = lg;
latitude = lt;
}
void show() {
cout << longitude << " ";
cout << latitude << "\n";
}
void *operator new(size_t size);
void operator delete(void *p);
void *operator new[](size_t size);
void operator delete[](void *p);
};
// new overloaded relative to loc.
void *loc::operator new(size_t size)
{
void *p;
cout << "In overloaded new.\n";
p = malloc(size);
if(!p) {
bad_alloc ba;
throw ba;
}
return p;
}
// delete overloaded relative to loc.
void loc::operator delete(void *p)
{
cout << "In overloaded delete.\n";
free(p);
}
// new overloaded for loc arrays.
void *loc::operator new[](size_t size)
{
void *p;
cout << "Using overload new[].\n";
p = malloc(size);
if(!p) {
bad_alloc ba;
throw ba;
}
return p;
}
// delete overloaded for loc arrays.
void loc::operator delete[](void *p)
```

```
{
cout << "Freeing array using overloaded delete[]\n";
free(p);
}
int main()
{
loc *p1, *p2;
int i;
try {
p1 = new loc (10, 20); // allocate an object
} catch (bad_alloc xa) {
cout << "Allocation error for p1.\n";
return 1;;
}
try {
p2 = new loc [10]; // allocate an array
} catch (bad_alloc xa) {
cout << "Allocation error for p2.\n";
return 1;;
}
p1->show();
for(i=0; i<10; i++)
p2[i].show();
delete p1; // free an object
delete [] p2; // free an array
return 0;
}
```

Overloading the nothrow Version of new and delete

You can also create overloaded nothrow versions of new and delete. To do so, use these skeletons.

```
/ Nothrow version of new.
void *operator new(size_t size, const nothrow_t &n)
{
// Perform allocation.
if(success) return pointer_to_memory;
else return 0;
}
// Nothrow version of new for arrays.
void *operator new[](size_t size, const nothrow_t &n)
{
// Perform allocation.
if(success) return pointer_to_memory;
else return 0;
```

```
}
void operator delete(void *p, const nothrow_t &n)
{
// free memory
}
void operator delete[](void *p, const nothrow_t &n)
{
// free memory
}
```

The type nothrow_t is defined in <new>. This is the type of the nothrow object. The nothrow_t parameter is unused.

Overloading Some Special Operators

C++ defines array subscripting, function calling, and class member access as operations.

Overloading [ ]

```
type class-name::operator[](int i)
{
// . . .
}
```

Given an object called O, the expression
O[3]
translates into this call to the operator[ ]( ) function: O.operator[](3)

Example

```
#include <iostream>
using namespace std;
class atype {
int a[3];
public:
```

```
atype(int i, int j, int k) {
a[0] = i;
a[1] = j;
a[2] = k;
}
int operator[](int i) { return a[i]; }
};
int main()
{
atype ob(1, 2, 3);
cout << ob[1]; // displays 2
return 0;
}
```

You can design the operator[ ]( ) function in such a way that the [ ] can be used on both the left and right sides of an assignment statement. To do this, simply specify the return value of operator[ ]( ) as a reference

```
#include <iostream>
using namespace std;
class atype {
int a[3];
public:
atype(int i, int j, int k) {
a[0] = i;
a[1] = j;
a[2] = k;
}
int &operator[](int i) { return a[i]; }
};
int main()
{
atype ob(1, 2, 3);
cout << ob[1]; // displays 2
cout << " ";
ob[1] = 25; // [] on left of =
cout << ob[1]; // now displays 25
return 0;
}
```

Because operator[ ]( ) now returns a reference to the array element indexed by i, it can be used on the left side of an assignment to modify an element of the array

- One advantage of being able to overload the [ ] operator is that it allows a means of implementing safe array indexing in C++. As you know, in C++, it is possible to overrun (or underrun) an array boundary at run time without generating a run-time error message

For example, this program adds a range check to the preceding program and proves that it works:

```
// A safe array example.
#include <iostream>
#include <cstdlib>
using namespace std;
class atype {
int a[3];
public:
atype(int i, int j, int k) {
a[0] = i;
a[1] = j;
a[2] = k;
}
int &operator[](int i);
};
// Provide range checking for atype.
int &atype::operator[](int i)
{
if(i<0 || i> 2) {
cout << "Boundary Error\n";
exit(1);
}
return a[i];
}
int main()
{
atype ob(1, 2, 3);

cout << ob[1]; // displays 2
cout << " ";
ob[1] = 25; // [] appears on left
cout << ob[1]; // displays 25
ob[3] = 44; // generates runtime error, 3 out-of-range
return 0;
}
```

Overloading ( )

When you overload the ( ) function call operator, you are not, per se, creating a new way to call a function. Rather, you are creating an operator function that can be passed an arbitrary number of parameters.

Syntax : double operator()(int a, float f, char *s);

and an object O of its class, then the statement
translates into this call to the operator( ) function.
O.operator()(10, 23.34, "hi");

```
#include <iostream>
using namespace std;
class loc {
int longitude, latitude;
public:
loc() {}
loc(int lg, int lt) {
longitude = lg;
latitude = lt;
}
void show() {
cout << longitude << " ";
cout << latitude << "\n";
}
loc operator+(loc op2);
loc operator()(int i, int j);
};
// Overload ( ) for loc.
loc loc::operator()(int i, int j)
{
longitude = i;
latitude = j;
return *this;
}
// Overload + for loc.
loc loc::operator+(loc op2)
{
loc temp;
temp.longitude = op2.longitude + longitude;
temp.latitude = op2.latitude + latitude;
return temp;
}
int main()
{
```

```
loc ob1(10, 20), ob2(1, 1);
ob1.show();
ob1(7, 8); // can be executed by itself
ob1.show();
ob1 = ob2 + ob1(10, 10); // can be used in expressions
ob1.show();
return 0;
}
```

The output produced by the program is shown here.

```
10 20
7 8
11 11
```

Overloading –>

he –> pointer operator, also called the class member access operator, is considered a unary operator when overloading

syntax : object->element;

```
#include <iostream>
using namespace std;
class myclass {
public:
int i;
myclass *operator->() {return this;}
};
int main()
{
myclass ob;
ob->i = 10; // same as ob.i
cout << ob.i << " " << ob->i;
return 0;
}
```

Overloading the Comma Operator

You can overload C++'s comma operator. The comma is a binary operator, and like all overloaded operators, you can make an overloaded comma perform any operation you want

```
#include <iostream>
using namespace std;
class loc {
```

```
int longitude, latitude;
public:
loc() {}
loc(int lg, int lt) {
longitude = lg;
latitude = lt;
}
void show() {
cout << longitude << " ";
cout << latitude << "\n";
}
loc operator+(loc op2);
loc operator,(loc op2);
};
// overload comma for loc
loc loc::operator,(loc op2)
{
loc temp;
temp.longitude = op2.longitude;
temp.latitude = op2.latitude;
cout << op2.longitude << " " << op2.latitude << "\n";
return temp;
}
// Overload + for loc
loc loc::operator+(loc op2)
{
loc temp;
temp.longitude = op2.longitude + longitude;
temp.latitude = op2.latitude + latitude;
return temp;
}
int main()
{
loc ob1(10, 20), ob2( 5, 30), ob3(1, 1);
ob1.show();
ob2.show();
ob3.show();
cout << "\n";
ob1 = (ob1, ob2+ob2, ob3);
ob1.show(); // displays 1 1, the value of ob3
return 0;
}
```

This program displays the following output:

```
10 20
5 30
1 1
```

10 60
1 1
1 1