Because classes have complicated internal structures, including data and functions, object initialization and cleanup for classes is much more complicated than it is for simple data structures. Constructors and destructors are special member functions of classes that are used to construct and destroy class objects.

* Construction may involve memory allocation and initialization for objects.
* Destruction may involve cleanup and deallocation of memory for objects.

Like other member functions, constructors and destructors are declared within a class declaration. They can be defined inline or external to the class declaration. Constructors can have default arguments. Unlike other member functions, constructors can have member initialization lists. The following restrictions apply to constructors and destructors:

* Constructors and destructors do not have return types nor can they return values.
* References and pointers cannot be used on constructors and destructors because their addresses cannot be taken.
* Constructors cannot be declared with the keyword **virtual**.
* Constructors and destructors cannot be declared **static**, **const**, or **volatile**.
* Unions cannot contain class objects that have constructors or destructors.

Constructors and destructors obey the same access rules as member functions. For example, if you declare a constructor with protected access, only derived classes and friends can use it to create class objects.

The compiler automatically calls constructors when defining class objects and calls destructors when class objects go out of scope. A constructor does not allocate memory for the class object its **this** pointer refers to, but may allocate storage for more objects than its class object refers to. If memory allocation is required for objects, constructors can explicitly call the **new** operator. During cleanup, a destructor may release objects allocated by the corresponding constructor. To release objects, use the **delete** operator.

Derived classes do not inherit constructors or destructors from their base classes, but they do call the constructor and destructor of base classes. Destructors can be declared with the keyword **virtual**.

Constructors are also called when local or temporary class objects are created, and destructors are called when local or temporary objects go out of scope.

#include <iostream>

using namespace std;

struct A {

virtual void f() { cout << "void A::f()" << endl; }

virtual void g() { cout << "void A::g()" << endl; }

virtual void h() { cout << "void A::h()" << endl; }

};

struct B : A {

virtual void f() { cout << "void B::f()" << endl; }

B() {

f();

g();

h();

}

};

struct C : B {

virtual void f() { cout << "void C::f()" << endl; }

virtual void g() { cout << "void C::g()" << endl; }

virtual void h() { cout << "void C::h()" << endl; }

};

int main() {

C obj;

}

The constructor of B does not call any of the functions overridden in C because C has been derived from B, although the example creates an object of type Cnamed obj.

You can use the **typeid** or the **dynamic\_cast** operator in constructors or destructors, as well as member initializers of constructors.

A *constructor* is a member function with the same name as its class. For example:

class X {

public:

X(); // constructor for class X

};

Constructors are used to create, and can initialize, objects of their class type.

You cannot declare a constructor as **virtual** or **static**, nor can you declare a constructor as **const**, **volatile**, or **const volatile**.

You do not specify a return type for a constructor. A return statement in the body of a constructor cannot have a return value.

A *default constructor* is a constructor that either has no parameters, or if it has parameters, *all* the parameters have default values.

If no user-defined constructor exists for a class A and one is needed, the compiler implicitly *declares* a constructor A::A(). This constructor is an inline public member of its class. The compiler will implicitly *define* A::A() when the compiler uses this constructor to create an object of type A. The constructor will have no constructor initializer and a null body.

The compiler first implicitly defines the implicitly declared constructors of the base classes and nonstatic data members of a class A before defining the implicitly declared constructor of A. No default constructor is created for a class that has any constant or reference type members.

A constructor of a class A is *trivial* if all the following are true:

* It is implicitly defined
* A has no virtual functions and no virtual base classes
* All the direct base classes of A have trivial constructors
* The classes of all the nonstatic data members of A have trivial constructors

If any of the above are false, then the constructor is *nontrivial*.

A union member cannot be of a class type that has a nontrivial constructor.

In simple words a "trivial" special member function literally means a member function that does its job in a very straightforward manner. The "straightforward manner" means different thing for different kinds of special member functions.

For a default constructor and destructor being "trivial" means literally "do nothing at all". For copy-constructor and copy-assignment operator, being "trivial" means literally "be equivalent to simple raw memory copying" (like copy with memcpy).

If you define a constructor yourself, it is considered non-trivial, even if it doesn't do anything, so a trivial constructor must be implicitly defined by the compiler.

In order for a special member function to satisfy the above requirements, the class must have a very simplistic structure, it must not require any hidden initializations when an object is being created or destroyed, or any hidden additional internal manipulations when it is being copied.

For example, if class has virtual functions, it will require some extra hidden initializations when objects of this class are being created (initialize virtual method table and such), so the constructor for this class will not qualify as trivial.

For another example, if a class has virtual base classes, then each object of this class might contain hidden pointers that point to other parts of the very same object. Such a self-referential object cannot be copied by a simple raw memory copy routine (like memcpy). Extra manipulations will be necessary to properly re-initialize the hidden pointers in the copy. For this reason the copy constructor and copy-assignment operator for this class will not qualify as trivial.

For obvious reasons, this requirement is recursive: all subobjects of the class (bases and non-static members) must also have trivial constructors.

Like all functions, a constructor can have default arguments. They are used to initialize member objects. If default values are supplied, the trailing arguments can be omitted in the expression list of the constructor. Note that if a constructor has any arguments that do not have default values, it is not a default constructor.

A *copy constructor* for a class A is a constructor whose first parameter is of type A&, const A&, volatile A&, or const volatile A&. Copy constructors are used to make a copy of one class object from another class object of the same class type. You cannot use a copy constructor with an argument of the same type as its class; you must use a reference. You can provide copy constructors with additional parameters as long as they all have default arguments. If a user-defined copy constructor does not exist for a class and one is needed, the compiler implicitly creates a copy constructor, with public access, for that class. A copy constructor is not created for a class if any of its members or base classes have an inaccessible copy constructor.

The following code fragment shows two classes with constructors, default constructors, and copy constructors:

class X {

public:

// default constructor, no arguments

X();

// constructor

X(int, int , int = 0);

// copy constructor

X(const X&);

// error, incorrect argument type

X(X);

};

class Y {

public:

// default constructor with one

// default argument

Y( int = 0);

// default argument

// copy constructor

Y(const Y&, int = 0);

};

A class object with a constructor must be explicitly initialized or have a default constructor. Except for aggregate initialization, explicit initialization using a constructor is the only way to initialize nonstatic constant and reference class members.

A class object that has no constructors, no virtual functions, no private or protected members, and no base classes is called an *aggregate*. Examples of aggregates are C-style structures and unions.

You explicitly initialize a class object when you create that object. There are two ways to initialize a class object:

* Using a parenthesized expression list. The compiler calls the constructor of the class using this list as the constructor's argument list.
* Using a single initialization value and the = operator. Because this type of expression is an initialization, not an assignment, the assignment operator function, if one exists, is not called. The type of the single argument must match the type of the first argument to the constructor. If the constructor has remaining arguments, these arguments must have default values.

The syntax for an initializer that explicitly initializes a class object with a constructor is:

Read syntax diagram[Skip visual syntax diagram](http://publib.boulder.ibm.com/infocenter/comphelp/v7v91/topic/com.ibm.vacpp7a.doc/language/ref/clrc15cplr387.htm#skipsyn-121)>>-+-(--*expression*--)-------------------+----------------------><

'-=--+-*expression*------------------+-'

| .-,----------. |

| V | |

'-{----*expression*-+--+---+--}-'

'-,-'

The following example shows the declaration and use of several constructors that explicitly initialize class objects:

// This example illustrates explicit initialization

// by constructor.

#include <iostream>

using namespace std;

class complx {

double re, im;

public:

// default constructor

complx() : re(0), im(0) { }

// copy constructor

complx(const complx& c) { re = c.re; im = c.im; }

// constructor with default trailing argument

complx( double r, double i = 0.0) { re = r; im = i; }

void display() {

cout << "re = "<< re << " im = " << im << endl;

}

};

int main() {

// initialize with complx(double, double)

complx one(1);

// initialize with a copy of one

// using complx::complx(const complx&)

complx two = one;

// construct complx(3,4)

// directly into three

complx three = complx(3,4);

// initialize with default constructor

complx four;

// complx(double, double) and construct

// directly into five

complx five = 5;

one.display();

two.display();

three.display();

four.display();

five.display();

}

The above example produces the following output:

re = 1 im = 0

re = 1 im = 0

re = 3 im = 4

re = 0 im = 0

re = 5 im = 0

**Initializing Base Classes and Members**

C++Constructors can initialize their members in two different ways. A constructor can use the arguments passed to it to initialize member variables in the constructor definition:

complx(double r, double i = 0.0) { re = r; im = i; }

Or a constructor can have an *initializer list* within the definition but prior to the function body:

complx(double r, double i = 0) : re(r), im(i) { /\* ... \*/ }

Both methods assign the argument values to the appropriate data members of the class.

The syntax for a constructor initializer list is:

Read syntax diagram[Skip visual syntax diagram](http://publib.boulder.ibm.com/infocenter/comphelp/v7v91/topic/com.ibm.vacpp7a.doc/language/ref/clrc15cplr388.htm#skipsyn-122) .-,---------------------------------------------------.

| .---------------------------. |

V V | |

>>-:----+-*identifier*-+--(----+-----------------------+-+--)-+--><

'-*class\_name*-' '-*assignment\_expression*-'

Include the initialization list as part of the function definition, not as part of the constructor declaration. For example:

#include <iostream>

using namespace std;

class B1 {

int b;

public:

B1() { cout << "B1::B1()" << endl; };

// inline constructor

B1(int i) : b(i) { cout << "B1::B1(int)" << endl; }

};

class B2 {

int b;

protected:

B2() { cout << "B2::B2()" << endl; }

// noninline constructor

B2(int i);

};

// B2 constructor definition including initialization list

B2::B2(int i) : b(i) { cout << "B2::B2(int)" << endl; }

class D : public B1, public B2 {

int d1, d2;

public:

D(int i, int j) : B1(i+1), B2(), d1(i) {

cout << "D1::D1(int, int)" << endl;

d2 = j;}

};

int main() {

D obj(1, 2);

}

The following is the output of the above example:

B1::B1(int)

B1::B1()

D1::D1(int, int)

If you do not explicitly initialize a base class or member that has constructors by calling a constructor, the compiler automatically initializes the base class or member with a default constructor. In the above example, if you leave out the call B2() in the constructor of class D (as shown below), a constructor initializer with an empty expression list is automatically created to initialize B2. The constructors for class D, shown above and below, result in the same construction of an object of class D:

class D : public B1, public B2 {

int d1, d2;

public:

// call B2() generated by compiler

D(int i, int j) : B1(i+1), d1(i) {

cout << "D1::D1(int, int)" << endl;

d2 = j;}

};

In the above example, the compiler will automatically call the default constructor for B2().

Note that you must declare constructors as public or protected to enable a derived class to call them. For example:

class B {

B() { }

};

class D : public B {

// error: implicit call to private B() not allowed

D() { }

};

The compiler does not allow the definition of D::D() because this constructor cannot access the private constructor B::B().

You must initialize the following with an initializer list: base classes with no default constructors, reference data members, non-static const data members, or a class type which contains a constant data member. The following example demonstrates this:

class A {

public:

A(int) { }

};

class B : public A {

static const int i;

const int j;

int &k;

public:

B(int& arg) : A(0), j(1), k(arg) { }

};

int main() {

int x = 0;

B obj(x);

};

The data members j and k, as well as the base class A must be initialized in the initializer list of the constructor of B.

You can use data members when initializing members of a class. The following example demonstrate this:

struct A {

int k;

A(int i) : k(i) { }

};

struct B: A {

int x;

int i;

int j;

int& r;

B(int i): r(x), A(i), j(this->i), i(i) { }

};

The constructor B(int i) initializes the following:

* B::r to refer to B::x
* Class A with the value of the argument to B(int i)
* B::j with the value of B::i
* B::i with the value of the argument to B(int i)

You can also call member functions (including virtual member functions) or use the operators **typeid** or **dynamic\_cast** when initializing members of a class. However if you perform any of these operations in a member initialization list before all base classes have been initialized, the behavior is undefined. The following example demonstrates this:

#include <iostream>

using namespace std;

struct A {

int i;

A(int arg) : i(arg) {

cout << "Value of i: " << i << endl;

}

};

struct B : A {

int j;

int f() { return i; }

B();

};

B::B() : A(f()), j(1234) {

cout << "Value of j: " << j << endl;

}

int main() {

B obj;

}

The output of the above example would be similar to the following:

Value of i: 8

Value of j: 1234

The behavior of the initializer A(f()) in the constructor of B is undefined. The run time will call B::f() and try to access A::i even though the base A has not been initialized.

The following example is the same as the previous example except that the initializers of B::B() have different arguments:

#include <iostream>

using namespace std;

struct A {

int i;

A(int arg) : i(arg) {

cout << "Value of i: " << i << endl;

}

};

struct B : A {

int j;

int f() { return i; }

B();

};

B::B() : A(5678), j(f()) {

cout << "Value of j: " << j << endl;

}

int main() {

B obj;

}

The following is the output of the above example:

Value of i: 5678

Value of j: 5678

The behavior of the initializer j(f()) in the constructor of B is well-defined. The base class A is already initialized when B::j is initialized.

**Construction Order of Derived Class Objects**

C++When a derived class object is created using constructors, it is created in the following order:

1. Virtual base classes are initialized, in the order they appear in the base list.
2. Nonvirtual base classes are initialized, in declaration order.
3. Class members are initialized in declaration order (regardless of their order in the initialization list).
4. The body of the constructor is executed.

The following example demonstrates this:

#include <iostream>

using namespace std;

struct V {

V() { cout << "V()" << endl; }

};

struct V2 {

V2() { cout << "V2()" << endl; }

};

struct A {

A() { cout << "A()" << endl; }

};

struct B : virtual V {

B() { cout << "B()" << endl; }

};

struct C : B, virtual V2 {

C() { cout << "C()" << endl; }

};

struct D : C, virtual V {

A obj\_A;

D() { cout << "D()" << endl; }

};

int main() {

D c;

}

The following is the output of the above example:

V()

V2()

B()

C()

A()

D()

The above output lists the order in which the C++ run time calls the constructors to create an object of type D.

## Destructors

C++*Destructors* are usually used to deallocate memory and do other cleanup for a class object and its class members when the object is destroyed. A destructor is called for a class object when that object passes out of scope or is explicitly deleted.

A destructor is a member function with the same name as its class prefixed by a ~ (tilde). For example:

class X {

public:

// Constructor for class X

X();

// Destructor for class X

~X();

};

A destructor takes no arguments and has no return type. Its address cannot be taken. Destructors cannot be declared **const**, **volatile**, **const volatile**or **static**. A destructor can be declared **virtual** or pure **virtual**.

If no user-defined destructor exists for a class and one is needed, the compiler implicitly declares a destructor. This implicitly declared destructor is an inline public member of its class.

The compiler will implicitly define an implicitly declared destructor when the compiler uses the destructor to destroy an object of the destructor's class type. Suppose a class A has an implicitly declared destructor. The following is equivalent to the function the compiler would implicitly define for A:

A::~A() { }

The compiler first implicitly defines the implicitly declared destructors of the base classes and nonstatic data members of a class A before defining the implicitly declared destructor of A

A destructor of a class A is *trivial* if all the following are true:

* It is implicitly defined
* All the direct base classes of A have trivial destructors
* The classes of all the nonstatic data members of A have trivial destructors

If any of the above are false, then the destructor is *nontrivial*.

A union member cannot be of a class type that has a nontrivial destructor.

Class members that are class types can have their own destructors. Both base and derived classes can have destructors, although destructors are not inherited. If a base class A or a member of A has a destructor, and a class derived from A does not declare a destructor, a default destructor is generated.

The default destructor calls the destructors of the base class and members of the derived class.

The destructors of base classes and members are called in the reverse order of the completion of their constructor:

1. The destructor for a class object is called before destructors for members and bases are called.
2. Destructors for nonstatic members are called before destructors for base classes are called.
3. Destructors for nonvirtual base classes are called before destructors for virtual base classes are called.

When an exception is thrown for a class object with a destructor, the destructor for the temporary object thrown is not called until control passes out of the catch block.

Destructors are implicitly called when an automatic object (a local object that has been declared **auto** or **register**, or not declared as **static** or **extern**) or temporary object passes out of scope. They are implicitly called at program termination for constructed external and static objects. Destructors are invoked when you use the **delete** operator for objects created with the **new** operator.

For example:

#include <string>

class Y {

private:

char \* string;

int number;

public:

// Constructor

Y(const char\*, int);

// Destructor

~Y() { delete[] string; }

};

// Define class Y constructor

Y::Y(const char\* n, int a) {

string = strcpy(new char[strlen(n) + 1 ], n);

number = a;

}

int main () {

// Create and initialize

// object of class Y

Y yobj = Y("somestring", 10);

// ...

// Destructor ~Y is called before

// control returns from main()

}

You can use a destructor explicitly to destroy objects, although this practice is not recommended. However to destroy an object created with the placement**new** operator, you can explicitly call the object's destructor. The following example demonstrates this:

#include <new>

#include <iostream>

using namespace std;

class A {

public:

A() { cout << "A::A()" << endl; }

~A() { cout << "A::~A()" << endl; }

};

int main () {

char\* p = new char[sizeof(A)];

A\* ap = new (p) A;

ap->A::~A();

delete [] p;

}

The statement A\* ap = new (p) A dynamically creates a new object of type A not in the free store but in the memory allocated by p. The statementdelete [] p will delete the storage allocated by p, but the run time will still believe that the object pointed to by ap still exists until you explicitly call the destructor of A (with the statement ap->A::~A()).

Nonclass types have a *pseudo destructor*. The following example calls the pseudo destructor for an integer type:

typedef int I;

int main() {

I x = 10;

x.I::~I();

x = 20;

}

The call to the pseudo destructor, x.I::~I(), has no effect at all. Object x has not been destroyed; the assignment x = 20 is still valid. Because pseudo destructors require the syntax for explicitly calling a destructor for a nonclass type to be valid, you can write code without having to know whether or not a destructor exists for a given type.

## Free Store

C++*Free store* is a pool of memory available for you to allocate (and deallocate) storage for objects during the execution of your program. The **new** and**delete** operators are used to allocate and deallocate free store, respectively.

You can define your own versions of **new** and **delete** for a class by overloading them. You can declare the **new** and **delete** operators with additional parameters. When **new** and **delete** operate on class objects, the class member operator functions **new** and **delete** are called, if they have been declared.

If you create a class object with the **new** operator, one of the operator functions **operator new()** or **operator new[]()** (if they have been declared) is called to create the object. An **operator new()** or **operator new[]()** for a class is always a static class member, even if it is not declared with the keyword **static**. It has a return type **void\*** and its first parameter must be the size of the object type and have type **std::size\_t**. It cannot be virtual.

Type **std::size\_t** is an implementation-dependent unsigned integral type defined in the standard library header <cstddef>. When you overload the **new**operator, you must declare it as a class member, returning type **void\***, with its first parameter of type **std::size\_t**. You can declare additional parameters in the declaration of **operator new()** or **operator new[]()**. Use the placement syntax to specify values for these parameters in an allocation expression.

The following example overloads two operator **new** functions:

* X::operator new(size\_t sz): This overloads the default **new** operator by allocating memory with the C function malloc(), and throwing a string (instead of std::bad\_alloc) if malloc() fails.
* X::operator new(size\_t sz, int location): This function takes an additional integer parameter, location. This function implements a very simplistic "memory manager" that manages the storage of up to three X objects.

Static array X::buffer holds three Node objects. Each Node object contains a pointer to an X object named data and a boolean variable namedfilled. Each X object stores an integer called number.

When you use this **new** operator, you pass the argument location which indicates the array location of buffer where you want to "create" your newX object. If the array location is not "filled" (the data member of filled is equal to false at that array location), the **new** operator returns a pointer pointing to the X object located at buffer[location].

#include <new>

#include <iostream>

using namespace std;

class X;

struct Node {

X\* data;

bool filled;

Node() : filled(false) { }

};

class X {

static Node buffer[];

public:

int number;

enum { size = 3};

void\* operator new(size\_t sz) throw (const char\*) {

void\* p = malloc(sz);

if (sz == 0) throw "Error: malloc() failed";

cout << "X::operator new(size\_t)" << endl;

return p;

}

void \*operator new(size\_t sz, int location) throw (const char\*) {

cout << "X::operator new(size\_t, " << location << ")" << endl;

void\* p = 0;

if (location < 0 || location >= size || buffer[location].filled == true) {

throw "Error: buffer location occupied";

}

else {

p = malloc(sizeof(X));

if (p == 0) throw "Error: Creating X object failed";

buffer[location].filled = true;

buffer[location].data = (X\*) p;

}

return p;

}

static void printbuffer() {

for (int i = 0; i < size; i++) {

cout << buffer[i].data->number << endl;

}

}

};

Node X::buffer[size];

int main() {

try {

X\* ptr1 = new X;

X\* ptr2 = new(0) X;

X\* ptr3 = new(1) X;

X\* ptr4 = new(2) X;

ptr2->number = 10000;

ptr3->number = 10001;

ptr4->number = 10002;

X::printbuffer();

X\* ptr5 = new(0) X;

}

catch (const char\* message) {

cout << message << endl;

}

}

The following is the output of the above example:

X::operator new(size\_t)

X::operator new(size\_t, 0)

X::operator new(size\_t, 1)

X::operator new(size\_t, 2)

10000

10001

10002

X::operator new(size\_t, 0)

Error: buffer location occupied

The statement X\* ptr1 = new X calls X::operator new(sizeof(X)). The statement X\* ptr2 = new(0) X calls X::operator new(sizeof(X),0).

The **delete** operator destroys an object created by the **new** operator. The operand of **delete** must be a pointer returned by **new**. If **delete** is called for an object with a destructor, the destructor is invoked before the object is deallocated.

If you destroy a class object with the **delete** operator, the operator function **operator delete()** or **operator delete[]()** (if they have been declared) is called to destroy the object. An **operator delete()** or **operator delete[]()** for a class is always a static member, even if it is not declared with the keyword **static**. Its first parameter must have type **void\***. Because **operator delete()** and **operator delete[]()** have a return type **void**, they cannot return a value.

The following example shows the declaration and use of the operator functions **operator new()** and **operator delete()**:

#include <cstdlib>

#include <iostream>

using namespace std;

class X {

public:

void\* operator new(size\_t sz) throw (const char\*) {

void\* p = malloc(sz);

if (p == 0) throw "malloc() failed";

return p;

}

// single argument

void operator delete(void\* p) {

cout << "X::operator delete(void\*)" << endl;

free(p);

}

};

class Y {

int filler[100];

public:

// two arguments

void operator delete(void\* p, size\_t sz) throw (const char\*) {

cout << "Freeing " << sz << " byte(s)" << endl;

free(p);

};

};

int main() {

X\* ptr = new X;

// call X::operator delete(void\*)

delete ptr;

Y\* yptr = new Y;

// call Y::operator delete(void\*, size\_t)

// with size of Y as second argument

delete yptr;

}

The above example will generate output similar to the following:

X::operator delete(void\*)

Freeing 400 byte(s)

The statement delete ptr calls X::operator delete(void\*). The statement delete yptr calls Y::operator delete(void\*, size\_t).

The result of trying to access a deleted object is undefined because the value of the object can change after deletion.

If **new** and **delete** are called for a class object that does not declare the operator functions **new** and **delete**, or they are called for a nonclass object, the global operators **new** and **delete** are used. The global operators **new** and **delete** are provided in the C++ library.

The C++ operators for allocating and deallocating arrays of class objects are **operator new[ ]()** and **operator delete[ ]()**.

You cannot declare the **delete** operator as virtual. However you can add polymorphic behavior to your **delete** operators by declaring the destructor of a base class as virtual. The following example demonstrates this:

#include <iostream>

using namespace std;

struct A {

virtual ~A() { cout << "~A()" << endl; };

void operator delete(void\* p) {

cout << "A::operator delete" << endl;

free(p);

}

};

struct B : A {

void operator delete(void\* p) {

cout << "B::operator delete" << endl;

free(p);

}

};

int main() {

A\* ap = new B;

delete ap;

}

The following is the output of the above example:

~A()

B::operator delete

The statement delete ap uses the **delete** operator from class B instead of class A because the destructor of A has been declared as virtual.

Although you can get polymorphic behavior from the **delete** operator, the **delete** operator that is statically visible must still be accessible even though another **delete** operator might be called. For example, in the above example, the function A::operator delete(void\*) must be accessible even though the example calls B::operator delete(void\*) instead.

Virtual destructors do not have any affect on deallocation operators for arrays (**operator delete[]()**). The following example demonstrates this:

#include <iostream>

using namespace std;

struct A {

virtual ~A() { cout << "~A()" << endl; }

void operator delete[](void\* p, size\_t) {

cout << "A::operator delete[]" << endl;

::delete [] p;

}

};

struct B : A {

void operator delete[](void\* p, size\_t) {

cout << "B::operator delete[]" << endl;

::delete [] p;

}

};

int main() {

A\* bp = new B[3];

delete[] bp;

};

The behavior of the statement delete[] bp is undefined.

When you overload the **delete** operator, you must declare it as class member, returning type **void**, with the first parameter having type **void\***, as described above. You can add a second parameter of type **size\_t** to the declaration. You can only have one **operator delete()** or **operator delete[]()** for a single class.

## Temporary Objects

C++It is sometimes necessary for the compiler to create temporary objects. They are used during reference initialization and during evaluation of expressions including standard type conversions, argument passing, function returns, and evaluation of the **throw** expression.

When a temporary object is created to initialize a reference variable, the name of the temporary object has the same scope as that of the reference variable. When a temporary object is created during the evaluation of a full-expression (an expression that is not a subexpression of another expression), it is destroyed as the last step in its evaluation that lexically contains the point where it was created.

There are two exceptions in the destruction of full-expressions:

* The expression appears as an initializer for a declaration defining an object: the temporary object is destroyed when the initialization is complete.
* A reference is bound to a temporary object: the temporary object is destroyed at the end of the reference's lifetime.

If a temporary object is created for a class with constructors, the compiler calls the appropriate (matching) constructor to create the temporary object.

When a temporary object is destroyed and a destructor exists, the compiler calls the destructor to destroy the temporary object. When you exit from the scope in which the temporary object was created, it is destroyed. If a reference is bound to a temporary object, the temporary object is destroyed when the reference passes out of scope unless it is destroyed earlier by a break in the flow of control. For example, a temporary object created by a constructor initializer for a reference member is destroyed on leaving the constructor.

In cases where such temporary objects are redundant, the compiler does not construct them, in order to create more efficient optimized code. This behavior could be a consideration when you are debugging your programs, especially for memory problems.

## User-Defined Conversions

C++*User-defined conversions* allow you to specify object conversions with constructors or with conversion functions. User-defined conversions are implicitly used in addition to standard conversions for conversion of initializers, functions arguments, function return values, expression operands, expressions controlling iteration, selection statements, and explicit type conversions.

There are two types of user-defined conversions:

* Conversion by constructor
* Conversion functions

The compiler can use only one user-defined conversion (either a conversion constructor or a conversion function) when implicitly converting a single value. The following example demonstrates this:

class A {

int x;

public:

operator int() { return x; };

};

class B {

A y;

public:

operator A() { return y; };

};

int main () {

B b\_obj;

// int i = b\_obj;

int j = A(b\_obj);

}

The compiler would not allow the statement int i = b\_obj. The compiler would have to implicitly convert b\_obj into an object of type A (withB::operator A()), then implicitly convert that object to an integer (with A::operator int()). The statement int j = A(b\_obj) explicitly convertsb\_obj into an object of type A, then implicitly converts that object to an integer.

User-defined conversions must be unambiguous, or they are not called. A conversion function in a derived class does not hide another conversion function in a base class unless both conversion functions convert to the same type. Function overload resolution selects the most appropriate conversion function. The following example demonstrates this:

class A {

int a\_int;

char\* a\_carp;

public:

operator int() { return a\_int; }

operator char\*() { return a\_carp; }

};

class B : public A {

float b\_float;

char\* b\_carp;

public:

operator float() { return b\_float; }

operator char\*() { return b\_carp; }

};

int main () {

B b\_obj;

// long a = b\_obj;

char\* c\_p = b\_obj;

}

The compiler would not allow the statement long a = b\_obj. The compiler could either use A::operator int() or B::operator float() to convertb\_obj into a **long**. The statement char\* c\_p = b\_obj uses B::operator char\*() to convert b\_obj into a **char\*** because B::operator char\*() hides A::operator char\*().

When you call a constructor with an argument and you have not defined a constructor accepting that argument type, only standard conversions are used to convert the argument to another argument type acceptable to a constructor for that class. No other constructors or conversions functions are called to convert the argument to a type acceptable to a constructor defined for that class. The following example demonstrates this:

class A {

public:

A() { }

A(int) { }

};

int main() {

A a1 = 1.234;

// A moocow = "text string";

}

The compiler allows the statement A a1 = 1.234. The compiler uses the standard conversion of converting 1.234 into an **int**, then implicitly calls the converting constructor A(int). The compiler would not allow the statement A moocow = "text string"; converting a text string to an integer is not a standard conversion.

## Copy Constructors

C++The *copy constructor* lets you create a new object from an existing one by initialization. A copy constructor of a class A is a non-template constructor in which the first parameter is of type A&, const A&, volatile A&, or const volatile A&, and the rest of its parameters (if there are any) have default values.

If you do not declare a copy constructor for a class A, the compiler will implicitly declare one for you, which will be an inline public member.

The following example demonstrates implicitly defined and user-defined copy constructors:

#include <iostream>

using namespace std;

struct A {

int i;

A() : i(10) { }

};

struct B {

int j;

B() : j(20) {

cout << "Constructor B(), j = " << j << endl;

}

B(B& arg) : j(arg.j) {

cout << "Copy constructor B(B&), j = " << j << endl;

}

B(const B&, int val = 30) : j(val) {

cout << "Copy constructor B(const B&, int), j = " << j << endl;

}

};

struct C {

C() { }

C(C&) { }

};

int main() {

A a;

A a1(a);

B b;

const B b\_const;

B b1(b);

B b2(b\_const);

const C c\_const;

// C c1(c\_const);

}

The following is the output of the above example:

Constructor B(), j = 20

Constructor B(), j = 20

Copy constructor B(B&), j = 20

Copy constructor B(const B&, int), j = 30

The statement A a1(a) creates a new object from a with an implicitly defined copy constructor. The statement B b1(b) creates a new object from b with the user-defined copy constructor B::B(B&). The statement B b2(b\_const) creates a new object with the copy constructor B::B(const B&, int). The compiler would not allow the statement C c1(c\_const) because a copy constructor that takes as its first parameter an object of type const C& has not been defined.

The implicitly declared copy constructor of a class A will have the form A::A(const A&) if the following are true:

* The direct and virtual bases of A have copy constructors whose first parameters have been qualified with **const** or **const volatile**
* The nonstatic class type or array of class type data members of A have copy constructors whose first parameters have been qualified with **const** or**const volatile**

If the above are not true for a class A, the compiler will implicitly declare a copy constructor with the form A::A(A&).

The compiler cannot allow a program in which the compiler must implicitly define a copy constructor for a class A and one or more of the following are true:

* Class A has a nonstatic data member of a type which has an inaccessible or ambiguous copy constructor.
* Class A is derived from a class which has an inaccessible or ambiguous copy constructor.

The compiler will implicitly define an implicitly declared constructor of a class A if you initialize an object of type A or an object derived from class A.

An implicitly defined copy constructor will copy the bases and members of an object in the same order that a constructor would initialize the bases and members of the object.

## Copy Assignment Operators

C++The *copy assignment operator* lets you create a new object from an existing one by initialization. A copy assignment operator of a class A is a nonstatic non-template member function that has one of the following forms:

* A::operator=(A)
* A::operator=(A&)
* A::operator=(const A&)
* A::operator=(volatile A&)
* A::operator=(const volatile A&)

If you do not declare a copy assignment operator for a class A, the compiler will implicitly declare one for you which will be inline public.

The following example demonstrates implicitly defined and user-defined copy assignment operators:

#include <iostream>

using namespace std;

struct A {

A& operator=(const A&) {

cout << "A::operator=(const A&)" << endl;

return \*this;

}

A& operator=(A&) {

cout << "A::operator=(A&)" << endl;

return \*this;

}

};

class B {

A a;

};

struct C {

C& operator=(C&) {

cout << "C::operator=(C&)" << endl;

return \*this;

}

C() { }

};

int main() {

B x, y;

x = y;

A w, z;

w = z;

C i;

const C j();

// i = j;

}

The following is the output of the above example:

A::operator=(const A&)

A::operator=(A&)

The assignment x = y calls the implicitly defined copy assignment operator of B, which calls the user-defined copy assignment operator A::operator=(const A&). The assignment w = z calls the user-defined operator A::operator=(A&). The compiler will not allow the assignment i = j because an operator C::operator=(const C&) has not been defined.

The implicitly declared copy assignment operator of a class A will have the form A& A::operator=(const A&) if the following are true:

* A direct or virtual base B of class A has a copy assignment operator whose parameter is of type const B&, const volatile B&, or B.
* A non-static class type data member of type X that belongs to class A has a copy constructor whose parameter is of type const X&, const volatile X&, or X.

If the above are not true for a class A, the compiler will implicitly declare a copy assignment operator with the form A& A::operator=(A&).

The implicitly declared copy assignment operator returns a reference to the operator's argument.

The copy assignment operator of a derived class hides the copy assignment operator of its base class.

The compiler cannot allow a program in which the compiler must implicitly define a copy assignment operator for a class A and one or more of the following are true:

* Class A has a nonstatic data member of a **const** type or a reference type
* Class A has a nonstatic data member of a type which has an inaccessible copy assignment operator
* Class A is derived from a base class with an inaccessible copy assignment operator.

An implicitly defined copy assignment operator of a class A will first assign the direct base classes of A in the order that they appear in the definition of A. Next, the implicitly defined copy assignment operator will assign the nonstatic data members of A in the order of their declaration in the definition of A.

Explicit constructors cannot take part in an implicit conversion.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | class Array  {  public:      Array(size\_t count);            // implicit ctor      explicit Array(size\_t count);  };    Array array = 1; // implicit conversion, ok    void UseArray(const Array& arr);    UseArray(10);    // mistake, want to avoid |

1. Why do we need implicit conversion? Sometimes it is more natural allow implicit conversion

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | class Number  {  public:      Number(int n);  };    Number num = 1233; // natural to do this |

1. When do you not need explicit declaration on a constructor? When constructor has more than one parameters.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | class UserName  {  public:      UserName(const string& first, const string& last);  };    UserName user("Arun", "Singh"); |

**References:**

<http://publib.boulder.ibm.com/infocenter/comphelp/v7v91/index.jsp?topic=%2Fcom.ibm.vacpp7a.doc%2Flanguage%2Fref%2Fclrc15cplr374.htm>

<http://abbeyhill.wordpress.com/2011/01/18/c-explicit-vs-implicit-constructors/>

Understanding Memory Layout

<http://eli.thegreenplace.net/2012/12/17/dumping-a-c-objects-memory-layout-with-clang/>