**Safe array**

In C++, there is no check to determine whether the array index is out of bounds. During program execution, an out-of-bound array index can cause serious problems. Also, recall that in C++ the array index starts at 0.

Safe array solves the out-of-bound array index problem and allows the user to begin the array index starting at any integer, positive or negative.

The following program shows how to overload the subscripting operator by using it to create

a "safe array" that prevents boundary errors. It defines a generic class called safe\_array,

which encapsulates an array. The type of the array is specified by a template type parameter

called T. The length of the array is specified by a non-type template parameter called len.

The array encapsulated by safe\_array is called ar. The length of the array is stored in a

variable called length. Both are private members of safe\_array. The array elements are

accessed only through the overloaded operator[ ]( ). It first confirms that an array access is

within bounds. If it is, operator[ ]( ) then returns a reference to the element. The length of the

array can be obtained by calling the method getlen( ).

// Overload [] to create a generic safe array type.

//

// The operator[]() function checks for array boundary errors

// so that an overrun or underrun is prevented.

//

// Notice that this example uses a non-type template parameter

// to specify the size of the array.

#include <iostream>

#include <cstdlib>

using namespace std;

// Here, T specifies the type of the array and the non-type

// parameter len specifies the length of the array.

template <class T, int len> class safe\_array {

// The array ar is declared to be of type T and of length len.

// The array is private. Access is allowed only through operator[]().

// In this way, boundary errors can be prevented.

T ar[len];

int length;

public:

// Create a safe\_array of type T with a length of len.

safe\_array();

// Overload the subscripting operator so that it accesses

// the elements in ar.

T &operator[](int i);

// Return the length of the array.

int getlen() { return length; }

};

// Create a safe\_array of type T with a length of len.

// The len variable is a non-type template parameter.

template <class T, int len> safe\_array<T, len>::safe\_array() {

// Initialize the array elements to their default value.

for(int i=0; i < len; ++i) ar[i] = T();

length = len;

}

// Return a reference to the element at the specified index.

// Provide range checking to prevent boundary errors.

template <class T, int len> T &safe\_array<T, len>::operator[](int i)

{

if(i < 0 || i > len-1) {

// Take appropriate action here. This is just

// a placeholder response.

cout << "\nIndex value of " << i << " is out-of-bounds.\n";

exit(1);

}

return ar[i];

}

// This is a simple class used to demonstrate an array of objects.

// Notice that the default constructor gives x the value -1.

class myclass {

public:

int x;

myclass(int i) { x = i; };

myclass() { x = -1; }

};

int main()

{safe\_array<int, 10> i\_ar; // integer array of size 10

safe\_array<double, 5> d\_ar; // double array of size 5

int i;

cout << "Initial values for i\_ar: ";

for(i=0; i < i\_ar.getlen(); ++i) cout << i\_ar[i] << " ";

cout << endl;

// Change the values in i\_ar.

for(i=0; i < i\_ar.getlen(); ++i) i\_ar[i] = i;

cout << "New values for i\_ar: ";

for(i=0; i < i\_ar.getlen(); ++i) cout << i\_ar[i] << " ";

cout << "\n\n";

cout << "Initial values for d\_ar: ";

for(i=0; i < d\_ar.getlen(); ++i) cout << d\_ar[i] << " ";

cout << endl;

// Change the values in d\_ar.

for(i=0; i < d\_ar.getlen(); ++i) d\_ar[i] = (double) i/3;

cout << "New values for d\_ar: ";

for(i=0; i < d\_ar.getlen(); ++i) cout << d\_ar[i] << " ";

cout << "\n\n";;

// safe\_array works with objects, too.

safe\_array<myclass, 3> mc\_ar; // myclass array of size 3

cout << "Initial values in mc\_ar: ";

for(i = 0; i < mc\_ar.getlen(); ++i) cout << mc\_ar[i].x << " ";

cout << endl;

// Give mc\_ar some new values.

mc\_ar[0].x = 19;

mc\_ar[1].x = 99;

mc\_ar[2].x = -97;

cout << "New values for mc\_ar: ";

for(i = 0; i < mc\_ar.getlen(); ++i) cout << mc\_ar[i].x << " ";

cout << endl;

// This creates a boundary overrun.

i\_ar[12] = 100;

// Comment-out the preceding line and then Uncomment the following

// line to generate a boundary underrun.

// i\_ar[-2] = 100;

return 0;

}The output is shown here:

Initial values for i\_ar: 0 0 0 0 0 0 0 0 0 0

New values for i\_ar: 0 1 2 3 4 5 6 7 8 9

Initial values for d\_ar: 0 0 0 0 0

New values for d\_ar: 0 0.333333 0.666667 1 1.33333

Initial values in mc\_ar: -1 -1 -1

New values for mc\_ar: 19 99 -97

Index value of 12 is out-of-bounds.

In the program, pay special attention to this statement:

i\_ar[12] = 100;

It attempts to assign 100 to location 12 within i\_ar. But i\_ar is only 10 elements long! If this

were a normal array, then a boundary overrun would occur. Fortunately, in this case, the

attempt is intercepted by operator[]( ) and the program is terminated before any damage

can be done. (In actual practice, some sort of error-handling would be supplied to deal with

the out-of-range condition; the program would not have to terminate.)

**A Safe Array that Uses Dynamic Allocation**

Before we begin, it is useful to contrast this approach with the one shown in Overload the

Subscripting Operator [ ] earlier . In that recipe, the example created an array

type called safe\_array that encapsulated a static array that actually held the elements. Thus,

each safe\_array was backed by a full-length static array. As a result, if a very large safe array

was needed, the resulting safe\_array object would also be very large because it would

encapsulate the entire array.

The version developed here uses a different approach. Called dyn\_safe\_array, it

dynamically allocates memory for the array and stores only a pointer to that memory. This

has the advantage of making the safe-array objects smaller—much smaller in some cases.

This makes them more efficient when they are passed to functions, for example. Of course,

it takes a bit more work to implement a safe array that uses dynamic memory, because both

a copy constructor and an overloaded assignment operator are needed. Like safe\_array

shown earlier, dyn\_safe\_array overloads the subscripting operator [ ] to allow normal,

array-like subscripting to access the elements in the array.

The dyn\_safe\_array class is generic, which means that it can be used to create any type of

array. The number of elements in the array is passed as a non-type argument in its template

specification. Its constructor then allocates sufficient memory to hold the array of the desired

size and type. A pointer to this memory is stored in aptr. The destructor for dyn\_safe\_array

automatically frees this memory when an object goes out of scope. Otherwise, because the [ ]

is overloaded, a dyn\_safe\_array can be used just like a normal array.

When one dyn\_safe\_array is used to initialize another, the copy constructor is called. It

creates a copy of the original object by first allocating memory for the array and then

copying elements from the original array into the newly allocated memory. This way, each

object's aptr points to its own array. Without the copy constructor, an identical copy of a

dyn\_safe\_array would be made, which would result in two objects having aptrs that

pointed to the same memory. Among other potential troubles, this would result in an

attempt to free the same memory more than once when the objects go out of scope. The

copy constructor prevents this.

The same type of problem that the copy constructor prevents can also occur when one

dyn\_safe\_array object is assigned to another. To avoid this problem, the assignment

operator is also overloaded so that the contents of the array are copied, but the dynamically

allocated memory used by each object remains separate.

One last point: The copy constructor and the overloaded assignment operator display

a message each time they are called. This is simply for illustration. Normally, neither would

generate any output.

// A generic safe-array class that prevents array boundary errors.

// It uses the subscripting operator to access the array elements.

// This version differs from the approach used in the recipe:

//

// Overload the Subscripting Operator []

//

// because it allocates memory for the array dynamically rather

// than statically.

//

// An explicit copy constructor is implemented so that a copy

// of a safe array object uses its own allocated memory. Therefore,

// the original object and the copy DO NOT point to the same

// memory. The assignment operator is also overloaded for the same

// reason. In both cases, the contents of the array are copied so

// that both the original array and the copy contain the same values.

#include <iostream>

#include <new>

#include <cstdlib>

using namespace std;

// A generic safe-array class that dynamically allocates memory

// for the array. The length of the array is passed as a non-type

// argument in the template specification.

template <class T, int len> class dyn\_safe\_array {

T \*aptr; // pointer to the memory that holds the array

int length; // number of elements in the array

public:

// The dyn\_safe\_array constructor.

dyn\_safe\_array();

// The dyn\_safe\_array copy constructor.

dyn\_safe\_array(const dyn\_safe\_array &obj);

// Release the allocated memory when a dyn\_safe\_array object

// goes out of scope.

~dyn\_safe\_array() {

delete [] aptr;

}

// Overload assignment.

dyn\_safe\_array &operator=(const dyn\_safe\_array<T,len> &rh\_op);

// Use the subscripting operator to access elements in

// the safe array.

T &operator[](int i);

// Return the size of the array.

int getlen() { return length; }

};

// This is dyn\_safe\_array's constructor.

template <class T, int len>

dyn\_safe\_array<T, len>::dyn\_safe\_array() {

try {

// Allocate the array.

aptr = new T[len];

} catch(bad\_alloc ba) {

cout << "Can't allocate array.\n";

// Take appropriate action here. This is just

// a placeholder response.

exit(1);

}

// Initialize the array elements to their default value.

for(int i=0; i < len; ++i) aptr[i] = T();

length = len;

}

// This is dyn\_safe\_array's copy constructor.

template <class T, int len>

dyn\_safe\_array<T, len>::dyn\_safe\_array(const dyn\_safe\_array &obj) {

cout << "Using dyn\_safe\_array's copy constructor to make a copy.\n";

try {

// Allocate an array of the same size as the

// one used by obj.

aptr = new T[obj.length];

} catch(bad\_alloc ba) {

// Take appropriate action here. This is just

// a placeholder response.

cout << "Can't allocate array.\n";

exit(1);

}

length = obj.length;

// Copy contents of the array.

for(int i=0; i < length; ++i)

aptr[i] = obj.aptr[i];

}

// Overload assignment so that a copy of the array is made.

// The copy is stored in an allocated memory that is separate

// from that used by the right-hand operand.

//

template<class T, int len> dyn\_safe\_array<T, len> & 474 H e r b S c h i l d t ' s C + + P r o g r a m m i n g C o o k b o o k 474 H e r b S c h i l d t ' s C + + P r o g r a m m i n g C o o k b o o k

dyn\_safe\_array<T, len>::operator=(const dyn\_safe\_array<T, len> &rh\_op) {

cout << "Assigning one dyn\_safe\_array object to another.\n";

// If necessary, release the memory currently used by the object.

if(aptr && (length != rh\_op.length)) {

// Delete the previously allocated memory.

delete aptr;

try {

// Allocate an array of the same size as the

// one used by rh\_op.

aptr = new T[rh\_op.length];

} catch(bad\_alloc ba) {

// Take appropriate action here. This is just

// a placeholder response.

cout << "Can't allocate array.\n";

exit(1);

}

}

length = rh\_op.length;

// Copy contents of the array.

for(int i=0; i < length; ++i)

aptr[i] = rh\_op.aptr[i];

return \*this;

}

// Provide range checking for dyn\_safe\_array by overloading

// the [] operator. Notice that a reference is returned.

// This lets an array element be assigned a value.

template <class T, int len> T &dyn\_safe\_array<T, len>::operator[](int i)

{

if(i < 0 || i > length) {

// Take appropriate action here. This is just

// a placeholder response.

cout << "\nIndex value of " << i << " is out-of-bounds.\n";

exit(1);

}

return aptr[i];

}

// A simple function for demonstration purposes.

// When called, the copy constructor will be used

// to create a copy of the argument passed to x.

template <class T, int len>

dyn\_safe\_array<T, len> f(dyn\_safe\_array<T, len> x) {

cout << "f() is returning a copy of x.\n";

return x;

}

C h a p t e r 7 : P o t p o u r r i 475 C h a p t e r 7 : P o t p o u r r i 475

// This is a simple class used to demonstrate an array of objects.

// Notice that the default constructor gives x the value -1.

class myclass {

public:

int x;

myclass(int i) { x = i; };

myclass() { x = -1; }

};

int main()

{

// Use the integer array.

dyn\_safe\_array<int, 5> i\_ar;

for(int i=0; i < i\_ar.getlen(); ++i) i\_ar[i] = i;

cout << "Contents of i\_ar: ";

for(int i=0; i < i\_ar.getlen(); ++i) cout << i\_ar[i] << " ";

cout << "\n\n";

// To generate a boundary overrun, uncomment the following line:

// i\_ar[19] = 10;

// To generate a boundary underrun, uncomment the following line:

// i\_ar[-2] = 10;

// Create a copy of i\_ar. This will invoke dyn\_safe\_array's copy constructor.

cout << "Create i\_ar2 and initialize it with i\_ar. This results\n"

<< "in dyn\_safe\_array's copy constructor being called.\n\n";

dyn\_safe\_array<int, 5> i\_ar2 = i\_ar;

cout << "Contents of i\_ar2: ";

for(int i=0; i < i\_ar2.getlen(); ++i) cout << i\_ar2[i] << " ";

cout << "\n\n";

// Create another safe array for integers, but don't assign

// it any values. This means that its elements will contain

// their default values.

cout << "Create i\_ar3.\n";

dyn\_safe\_array<int, 5> i\_ar3;

cout << "Original contents of i\_ar3: ";

for(int i=0; i < i\_ar3.getlen(); ++i) cout << i\_ar3[i] << " ";

cout <<"\n\n";

// Now, pass i\_ar3 to f() and assign the result to i\_ar:

cout << "Now, this line will execute: i\_ar3 = f(i\_ar);\n"

<< "This will result in the following sequence of events:\n"

<< " 1. dyn\_safe\_array's copy constructor is called to make a\n"

<< " copy of i\_ar that is passed to the x parameter of f().\n"

<< " 2. The copy constructor is called again when a copy\n"

<< " is made for the return value of f().\n"

<< " 3. The overloaded assignment operator is called to\n"

<< " assign the result of f() to i\_ar3.\n\n";

i\_ar3 = f(i\_ar);

476 H e r b S c h i l d t ' s C + + P r o g r a m m i n g C o o k b o o k 476 H e r b S c h i l d t ' s C + + P r o g r a m m i n g C o o k b o o k

cout << "Contents of i\_ar3 after receiving value from f(i\_ar): ";

for(int i=0; i < i\_ar3.getlen(); ++i) cout << i\_ar3[i] << " ";

cout << "\n\n";

cout << "Of course, dyn\_safe\_array works with class types, too.\n";

dyn\_safe\_array<myclass, 3> mc\_ar;

cout << "Original contents of mc\_ar: ";

for(int i=0; i < mc\_ar.getlen(); ++i) cout << mc\_ar[i].x << " ";

cout << endl;

mc\_ar[0].x = 9;

mc\_ar[1].x = 8;

mc\_ar[2].x = 7;

cout << "Values in mc\_ar after setting them: ";

for(int i=0; i < mc\_ar.getlen(); ++i) cout << mc\_ar[i].x << " ";

cout << "\n\n";

cout << "Now, create mc\_ar2 and then execute this statement:\n"

<< " mc\_ar2 = f(mc\_ar);\n\n";

dyn\_safe\_array<myclass, 3> mc\_ar2;

mc\_ar2 = f(mc\_ar);

cout << "Contents of mc\_ar2 after receiving f(mc\_ar): ";

for(int i=0; i < mc\_ar2.getlen(); ++i) cout << mc\_ar2[i].x << " ";

cout << endl;

return 0;

}

The output is shown here:

Contents of i\_ar: 0 1 2 3 4

Create i\_ar2 and initialize it with i\_ar. This results

in dyn\_safe\_array's copy constructor being called.

Using dyn\_safe\_array's copy constructor to make a copy.

Contents of i\_ar2: 0 1 2 3 4

Create i\_ar3.

Original contents of i\_ar3: 0 0 0 0 0

Now, this line will execute: i\_ar3 = f(i\_ar);

This will result in the following sequence of events:

1. dyn\_safe\_array's copy constructor is called to make a

copy of i\_ar that is passed to the x parameter of f().

2. The copy constructor is called again when a copy

is made for the return value of f().

3. The overloaded assignment operator is called to

assign the result of f() to i\_ar3.

Using dyn\_safe\_array's copy constructor to make a copy.

f() is returning a copy of x.

Using dyn\_safe\_array's copy constructor to make a copy.

Assigning one dyn\_safe\_array object to another. C h a p t e r 7 : P o t p o u r r i 477 C h a p t e r 7 : P o t p o u r r i 477

Contents of i\_ar3 after receiving value from f(i\_ar): 0 1 2 3 4

Of course, dyn\_safe\_array works with class types, too.

Original contents of mc\_ar: -1 -1 -1

Values in mc\_ar after setting them: 9 8 7

Now, create mc\_ar2 and then execute this statement:

mc\_ar2 = f(mc\_ar);

Using dyn\_safe\_array's copy constructor to make a copy.

f() is returning a copy of x.

Using dyn\_safe\_array's copy constructor to make a copy.

Assigning one dyn\_safe\_array object to another.

Contents of mc\_ar2 after receiving f(mc\_ar): 9 8 7

Options and Alternatives

As explained in the discussion, the most common form of copy constructor has only one

parameter that is a reference to an object of the class for which the copy constructor is

defined. However, it is permissible for a copy constructor to have additional parameters as

long as they have default arguments. For example, assuming the dyn\_safe\_array class, the

following declaration specifies a valid copy constructor:

dyn\_safe\_array(const dyn\_safe\_array &obj, int num = -1);

Here, the num parameter defaults to –1. You could use this constructor to allow only the

first num elements of the new dyn\_safe\_array to be initialized by the first num elements of

obj. The remaining elements can be given a default value. When num is –1, the entire array

is initialized by obj. This version of the copy constructor could be written like this:

// If num is not –1, initialize the first num elements of a safe array

// using the value from obj. The remaining elements get default values.

// Otherwise, initialize the entire array with the elements from obj.

template <class T, int len>

dyn\_safe\_array<T, len>::dyn\_safe\_array(const dyn\_safe\_array &obj,

int num) {

cout << "Using dyn\_safe\_array's copy constructor to make a copy.\n";

try {

// Allocate an array of the same size as the

// one used by obj.

aptr = new T[obj.length];

} catch(bad\_alloc ba) {

// Take appropriate action here. This is just

// a placeholder response.

cout << "Can't allocate array.\n";

exit(1);

}

length = obj.length;

// Copy contents of obj, up to the number passed via num. 478 H e r b S c h i l d t ' s C + + P r o g r a m m i n g C o o k b o o k 478 H e r b S c h i l d t ' s C + + P r o g r a m m i n g C o o k b o o k

// If num is -1, then all values are copied.

if(num == -1) num = obj.length;

for(int i=0; i < num; ++i)

aptr[i] = obj.aptr[i];

// Initialize any remaining elements with their default value.

for(int i=num; i < length; ++i)

aptr[i] = T();

}

You could use this constructor as shown here:

dyn\_safe\_array<int, 5> i\_ar2(i\_ar, 3);

Here, the first three elements of i\_ar are used to initialize the first three elements of i\_ar2.

The remaining elements are given a default value, which for integers, is zero.

As explained in the discussion (and demonstrated by the dyn\_safe\_array class in the Example), if you need to implement a copy constructor, you often also need to

overload the assignment operator. The reason is that the same issues that necessitate the

copy constructor will also be present during assignment. It is important to not overlook

assignment.

**Alternative example:**

#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) {

**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];

}

**void** put(**int** i, **int** j) {

**if**(i>=0 && i<size) p[i] = j;

}

**int** get(**int** i) {

**return** p[i];

}

};

**int** main()

{

array num(10);

**for**(**int** i=0; i<10; i++)

num.put(i, i);

**for**(**int** i=9; i>=0; i--)

cout << num.get(i);

cout << "\n";

array x(num); // invokes copy constructor

**for**(**int** i=0; i<10; i++)

cout << x.get(i);

**return** 0;

}