How to use pointers to work with dynamic memory

One of the most common uses of pointers is to allocate *dynamic memory*, which is memory that's allocated at runtime. Now, you'll learn about the different types of memory. Then, you'll learn how to use pointers to work with dynamic memory.

An overview of the types of storage

When a program starts, the system sets aside memory for that program. Four of the types of memory, also known as *storage*, are described at the top of figure 17-9.

The first three types of storage are allocated for specific purposes. *Code storage*, as its name implies, is memory that's allocated for the program code itself. *Static storage* is memory that's allocated for global variables, like the ones you learned about in chapter 7. And *automatic storage* is memory that's allocated for functions and local variables.

With automatic storage, the system automatically allocates memory when you create an object and automatically deallocates it when the object goes out of scope. To manage the local variables and functions, automatic storage uses a last-in-first-out (LIFO) stack. That's why automatic storage is also known as *stack storage*, *stack memory*, the *call stack*, or the *stack*.

One of the benefits of automatic storage is that you can use it just by declaring a variable. Another benefit is that it can be accessed quickly.

One of the drawbacks of automatic storage is that it can be inflexible. For instance, you must know the size of a built-in array at compile time. Another drawback is that the amount of available stack storage is limited. If you try to use more than what's available, you'll get a *stack overflow* error. For example, this error can occur if you define a built-in array that's larger than the memory the stack has available.

Free store storage, or the *free store*, is the remaining memory that's set aside for a program but that's not yet allocated. This type of storage is also called *heap memory* or the *heap*.

One of the benefits of the free store is that there's much more of it than there is stack memory. Another benefit is that it's flexible. In fact, the free store is what gives the vector and string objects their flexibility and the ability to hold large amounts of data.

One of the drawbacks of the free store is that it can't be accessed as quickly as automatic storage. Another drawback is that the programmer must add code to manually allocate and deallocate memory. If the programmer fails to properly deallocate free store memory, it can lead to memory leaks or heap corruption. You'll learn more about that later in this chapter.

Types of storage set aside for a program when it starts

Name	Description
Code storage	Memory allocated for the program code itself.
Static storage	Memory allocated for global variables.
Automatic storage	Memory allocated for local variables and functions. This memory uses a last-in-first-out (LIFO) *stack* to manage local variables and functions. As a result, it's also called *stack storage*, *stack memory*, the *call stack*, or the *stack*.
Free store	The remaining, unallocated memory. This memory is also called *heap memory* or the *heap*.

Automatic storage (stack)

Benefits

- Access is fast.
- Allocation and deallocation of memory is handled by the system automatically.

Drawbacks

- Less flexible. For instance, the size of an array must be known at compile time.
- The memory allocated for the stack is limited, so you can run out of automatic storage. Running out of stack memory is known as *stack overflow*.

Free store (heap)

Benefits

- More flexible.
- Much more available memory.

Drawbacks

- Access is slower.
- Allocation and deallocation of memory must be handled manually by the program code. Failure to properly deallocate free store memory can lead to *memory leaks* or *heap corruption*.

Description

- When a program starts, the system sets aside memory for the program code, global and local variables, and functions. It also sets aside memory that can be allocated as the program runs.
- *Dynamic memory* is memory that's allocated at runtime. The free store is a type of dynamic memory.
- Unlike stack memory, free store memory is *not* automatically deallocated when the code goes out of scope. Instead, the program must manually deallocate it.

Figure 17-9 An overview of the types of storage

How to allocate and deallocate free store memory

To work with free store memory, you use the new and delete keywords. The new keyword allocates memory on the free store and returns a pointer to the location where the memory is allocated. The delete keyword deallocates that memory and returns it to the free store so it can be used again.

The example in figure 17-10 shows how this works. Here, the first two statements show how to allocate a single object. To start, the first statement uses the new keyword to allocate memory for a double object and return a pointer to it. This assigns that pointer to the variable named pi. The second statement works similarly, but it includes an initial value for the double object that's created.

The next two statements show how to allocate memory to an array of objects. Here, the first statement uses the new keyword to create an array of 10 doubles and return a pointer to it that's assigned to the variable named arr. The second statement works similarly, but it includes a list of initial values for the array.

As you learned in chapter 12, a built-in array on the stack decays to a pointer. This pointer points to the first element in the array and doesn't know how many elements the array contains. A pointer to an array on the heap works similarly, except that it can be allocated at runtime. As a result, you don't have to code the size of the array as a constant.

Once you use the new keyword to allocate objects on the free store, you need to use the delete keyword to deallocate them. This is important because, unlike automatic storage, an object that's allocated on the free store is *not* automatically deallocated when it goes out of scope. Instead, you must write code to deallocate the memory yourself.

The second example shows how to use the delete keyword to deallocate memory. Here, the first statement uses the delete keyword to deallocate the memory for the object that the pointer named pi points to. Then, the second statement uses the delete keyword followed by square brackets (`[]`) to deallocate the memory for the array that the pointer named arr points to.

Although this seems easy enough, in practice, using the new and delete keywords can cause problems. For example, if you forget to call delete, the memory is never returned to the free store and can't be used again. This is known as a *memory leak* and it's shown by the third example. If the program runs for a short time, it may not matter because all memory is returned when a program ends. But for long-running programs, memory leaks can build up and degrade performance over time.

Unfortunately, you can't fix a memory leak just by coding the delete keyword throughout your code. That's because attempting to deallocate memory that's already been deallocated can corrupt that memory. This is known as *heap corruption*, and it's shown by the fourth example.

If you work with legacy C or C++ code, you may find that it uses the malloc and free keywords to allocate and deallocate memory. This is an older technique that's not considered a best practice for new development, but you might need to use it to maintain old code. If you do that, however, you should make sure not to use the malloc/free keywords and the new/delete keywords in the same program as that can lead to serious problems.

How to use the new keyword to allocate free store memory

Allocate a single object of type double

```
double* pi = new double;
double* price = new double(5.99);      // returns pointer to double
                                       // include initial value
```

Allocate a dynamic array of doubles

```
double* arr = new double[10];          // returns pointer to first element
double* prices = new double[3]{5.99,6.99,7.99};
                                       // include initial values
```

How to use the delete keyword to deallocate free store memory

Deallocate a single object

```
delete pi;
```

Deallocate an array

```
delete[] arr;
```

A function that could lead to a memory leak

```
int* create_array(unsigned int size) {
    return new int[size]{0};  // returns pointer to array on free store
}
```

This code neglects to call delete[ ] on the pointer returned by the function

```
void main() {
    int* arr = create_array(20);
    /* code that works with the array goes here */
}
```

A function that could lead to heap corruption

```
void display_array(int* arr, unsigned int size) {
    for (int i = 0; i < size; ++i) {
        cout << arr[i] << ' ';
    }
    delete[] arr;              // deletes array when done
}
```

This code calls delete on a pointer to stack memory

```
int arr[5]{0};                 // allocate memory on stack
display_array(arr, 5);         // function calls delete on stack memory
```

This code calls delete twice on a pointer to heap memory

```
int* arr = new int[5]{0}; // allocate memory on heap
display_array(arr, 5);    // function calls delete on heap memory
delete[] arr;             // statement calls delete on heap memory again
```

Description

- You can use the new keyword to allocate free store memory. This keyword returns a pointer to the newly allocated memory.
- You can use the delete keyword to deallocate a single object, and you can use the delete keyword followed by square brackets (`[]`) to deallocate an array.

Figure 17-10 How to allocate and deallocate free store memory

How to avoid memory leaks and corruption

When you allocate and deallocate memory, it's possible to write code that leaks or corrupts memory. The topics that follow present two techniques that can help you avoid these problems.

How to use RAII (Resource Acquisition Is Instantiation)

When you define an object on the stack, the system automatically calls its constructor to create it. Then, when that stack object goes out of scope, the system automatically calls its destructor to destroy it.

RAII (*Resource Acquisition Is Instantiation*) is a programming pattern that takes advantage of this process to safely allocate and deallocate free store memory used by an object. Basically, RAII states that you should allocate free store memory in the constructor of an object and deallocate that memory in the destructor of that object. That way, the free store memory is automatically managed by the processes of stack storage, and the programmer can just use the object without needing to manually manage the free store memory that it uses.

Figure 17-11 defines a class named MyContainer that shows how this works. To start, this class defines two private data members, a pointer to an int variable named elements and an int variable named size. This class also defines public constructor and destructor functions.

Within the constructor, the new keyword allocates an array on the free store. This returns a pointer to the dynamic array that's assigned to the data member named elements. Then, this constructor displays a message on the console that indicates that the memory has been allocated.

Within the destructor, the delete[] keyword deallocates the free store memory that was allocated in the constructor. Then, this destructor displays a message on the console that indicates that the memory has been deallocated.

This illustrates how an object created from the MyContainer class handles its free store memory cleanly. When you create a MyContainer object, it allocates memory for an int array with ten elements. In other words, a *resource* is *acquired* when the object is *instantiated*. Then, when that object goes out of scope, the system calls the destructor, which deallocates the memory.

In the second example, for instance, free store memory is allocated when the main() function defines a MyContainer object. Then, when the main() function ends and this object goes out of scope, the memory is automatically deallocated.

Because RAII doesn't mention freeing the resource, some programmers have proposed more descriptive names like Constructor Acquires Destructor Releases (CADRe) or Scope-Based Resource Management (SBRM). However, RAII is the most common name for this pattern.

When using RAII, you should know that some C++ compilers don't automatically call destructors when an exception occurs. This can keep your automatic deallocation from occurring. To correct for this, you can put all code in the main() function within a try/catch block that handles all exceptions.

A MyContainer class that uses RAII

```
class MyContainer {
private:
    int* elements;
    int size = 10;
public:
    // constructor
    MyContainer() {
        elements = new int[size];                          // allocate memory
        cout << "memory allocated for MyContainer object\n";
    }

    // destructor
    ~MyContainer() {
        delete[] elements;
        cout << "memory deallocated for MyContainer object\n";  // deallocate memory
    }
};
```

Code that uses the MyContainer class

```
int main() {
    cout << "main() function starting...\n";
    MyContainer mine;
    cout << "main() function ending...\n";
}
```

The console

```
main() function starting...
memory allocated for MyContainer object
main() function ending...
memory deallocated for MyContainer object
```

Description

- When you create an object on the stack, the object's constructor runs when the object is created and its destructor runs when it goes out of scope.
- *RAII* (*Resource Acquisition Is Instantiation*) is a programming pattern that takes advantage of the way the stack creates and destroys objects to allocate and deallocate free store memory automatically.
- For RAII to work, the object must be created on the stack. If the object is created on the heap, its destructor won't be called when it goes out of scope.
- RAII can also be used to work with other system resources such as file handles.
- Some C++ compilers don't call destructors (sometimes called *unwinding the stack*) when an unhandled exception is thrown. To account for this, you can code a try/catch statement around all code in the main() function of your program that handles all exceptions.

Figure 17-11 How to use RAII (Resource Acquisition Is Instantiation)