Chapter 10 C++ Dynamic Memory

Objectives

- To understand the similarities and differences between C++ pointers and Python references.
- To learn how to use the C++ operators that access memory addresses and dereference pointers.
- To understand how to dynamically allocate and deallocate memory in C++.
- To learn how to write classes in C++ that allocate and deallocate dynamic memory.

10.1 Introduction

As we briefly discussed in earlier chapters, the internal mechanisms that Python and C++ use for storing data in variables and names are different. In this chapter we will discuss these differences in detail. C++'s default mechanism for storing variables is different than Python's, but C++ does support pointer variables that are similar to Python references. C++ programmers can choose which mechanism to use depending on the efficiency and capabilities they need. C++ pointers give us the flexibility to delay memory allocation decisions until run-time. This makes it possible to change the size of arrays at run-time and create linked structures in C++. Using C++ pointers does require much more care than using Python references; it is easy to make mistakes with pointers and create a program that gives unexpected results or crashes. This chapter and the next chapter will cover the use of dynamic memory and pointers. We will begin by reviewing the basic memory models of Python and C++.

Python names are a reference to a memory location where the actual data is stored along with type information and a reference count; different names can refer to the same data object and assignment statements make the name refer to a different data object. C++ associates (binds) a memory location with each variable and the same memory location is used for that variable throughout the lifetime of the variable. Each assignment statement causes different data to be stored in the memory location bound to the variable. Here is a C++ example:

```
// memory.cpp
#include <iostream>
using namespace std;

int main()
{
  int x, y, z;
  x = 3;
  y = 4;
  z = x;
  x = y;
  cout << x << " " << y << " " << z << endl;
  return 0;
}
```

The following table shows a representation of memory while this program is executing. When the `main` function begins execution, four bytes are allocated for each of the three integers. We have started our table at the memory location `1000`, but the specific memory address used is not important and can vary each time the program is run. The key point to notice is that the memory location used for each variable does not change; the data stored at the memory location does change as different values are assigned to the variable. As you would expect, the program outputs `4 4 3`.

Memory address	**Variable name**	**Data value**
1000	x	3 then 4
1004	y	4
1008	z	3

The Python version of this program is the following:

```
# memory.py
x = 3
y = 4
z = x
x = y
print x, y, z
```

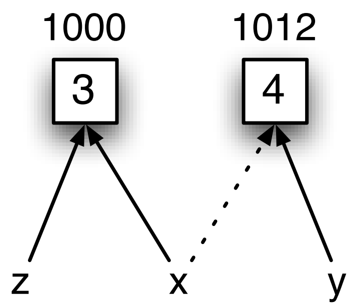

Figure 10.1: Picture of Python memory references

The end result of executing comparable code for C++ variables and Python references to immutable types is the same even though the internal representations are different. The Python program also outputs `4 4 3`. Figure 10.1 shows a pictorial representation of the memory for this Python code. The key point to notice here is that there is one copy of the `3` object and one copy of the `4` object at fixed memory locations; the names refer to these objects and as the code executes, the memory location that `x` refers to changes from `1000` to `1012`. At the end, we have multiple names referring to the same memory location. We have not shown the reference count and type information for the object, but each Python integer object requires 12 bytes on 32-bit systems.

The important differences between Python and C++ are

- Each C++ variable corresponds to a fixed memory location where data is stored; each time a value is assigned to that variable, the same memory location is used to store the data.
- A Python name refers to an object in memory. Python objects must also store information about their type and a reference count, so storing data in Python requires more space than storing the same data in C++.

- Assigning a Python name to an object changes the reference so that it refers to a different object (i.e., the memory address the name refers to changes).
- It is possible to have multiple names refer to the same object in Python. Modifying a mutable object via one name affects the other name since both names refer to the same object. In C++, each variable gets its own fixed address so changing one variable does not affect other variables.
- C++ does support references, but they are not commonly used. C++ also supports pointers, which are commonly used and allow us to perform similar types of operations that Python references support.

The differences between using references in Python and storing the actual data in variables in C++ become apparent when you modify a mutable type instead of assigning a variable to a new or different object. We will use our `Rational` class presented in earlier chapters to demonstrate this. The corresponding Python and C++ code fragments we will examine are

```
# Python code
r1 = Rational()
r1.set(2, 3)
r2 = r1
r1.set(1, 3)
print r1
print r2
// C++ code
Rational r1, r2;
r1.set(2, 3);
r2 = r1;
r1.set(1, 3);
cout << r1 << endl;
cout << r2 << endl;
```

These assume we have defined the appropriate methods for our classes. The `set` method in Python and C++ is

```
# Python code
    def set(self, num, den):
        self.num = num
        self.den = den
// C++ code
void Rational::set(int n, int d)
{
  num_ = n;
  den_ = d;
}
```

In Python, both `r1` and `r2` refer to the same object that we will assume is stored at memory location `1000`, as the following table shows. Additional memory is required in Python to store information about the data type for the object and the reference count for the object, but we will not include that here. Since we are creating a `Rational` object that has the instance variables `num` and `den`, we have named the variable name column based on the instance variable names of the `Rational` object.

Memory address	Variable name	Data value
1000	`.num`	reference to 2, then 1
1004	`.den`	reference to 3

If you recall our discussions from section 4.2, you will understand that the sample Python code will change the one `Rational` object to which both names refer.

The corresponding C++ declaration of `r1` and `r2` results in two `Rational` objects being created, requiring a total of 16 bytes of memory being allocated as the following table shows. Each `Rational` object requires eight bytes since it has two `int` instance variables that each require four bytes. No additional memory is needed in C++ since the C++ run-time environment does not need to keep track of the data type or the reference count.

Memory address	Variable name	Data value
1000	`r1.num`	?
1004	`r1.den`	?
1008	`r2.num`	?
1012	`r2.den`	?

After the statement `r1.set(2, 3)`, the memory now holds

Memory address	Variable name	Data value
1000	`r1.num`	2
1004	`r1.den`	3
1008	`r2.num`	?
1012	`r2.den`	?

Unless we have defined our own `operator=` (we will discuss this in subsection 10.4.3) for the C++ `Rational` class, the execution of `r2 = r1` effectively causes the two statements `r2.num = r1.num` and `r2.den = r1.den` to be executed. We cannot explicitly write those two statements ourselves since the instance variables are private, but the compiler generates the code that performs those two assignments. This causes the data stored at memory location 1000 to be copied to memory location 1008 and the data stored at memory location 1004 to be copied to memory location 1012. The following table shows the memory representation after the assignment statement.

Memory address	Variable name	Data value
1000	`r1.num`	2
1004	`r1.den`	3
1008	`r2.num`	2
1012	`r2.den`	3

After the statement `r1.set(1, 3)`, the memory now holds

Memory address	Variable name	Data value
1000	`r1.num`	1
1004	`r1.den`	3
1008	`r2.num`	2
1012	`r2.den`	3

Unlike in the Python version, `r1` and `r2` now hold different values so the output for `r1` is `1/3` and the output for `r2` is `2/3`. If we had instead executed the statement `r2.set(1, 3)`, the output for `r1` would be `2/3` and the output for `r2` would be `1/3`. The difference between Python and C++ is that each declared C++ variable gets its own memory location to store the instance variables and assigning one variable to another copies the data, but assigning one Python name to another results in both names referring to the same object.

These different mechanisms for managing memory have trade-offs. Python's mechanism allows dynamic typing and supports linked structures. However, overall Python uses more memory since we have to store the identifier name in a dictionary, the references, and the actual data with type information and a reference count. It also requires two memory accesses to get the data for a given Python name. C++'s mechanism uses less memory, and is almost always faster. One case where Python is faster is assignment of two names that are a class with a large amount of data. C++ effectively gives a deep copy while Python makes a reference to the same data. So assignment is faster in Python for class objects, but is not performing an equivalent operation. This is the reason variables that are instances of a class are typically passed by reference in C++ even when we do not want to change the data for the variable. As we discussed, the `const` designation is used when we do not want to change the data so the compiler will make certain our code does not change it. Computer scientists use the term *reference semantics* to describe how Python's assignment statement works, since it creates another reference, and use the term *value semantics* to describe how C++'s assignment statement works, since it copies the value of the variable.

Python's memory management mechanism is known as *implicit heap dynamic*. The Python interpreter automatically allocates and deallocates memory as needed. A section of memory known as a *dynamic memory heap* (sometimes referred to simply as a *heap*) is used for these allocations and deallocations. The default C++ memory management mechanism is known as *stack dynamic*. When a function is called, the amount of space needed for the variables is allocated on a stack. Since in most cases we can determine at compile time how much memory is needed for all the local variables, one machine language instruction can be used to allocate the space on the stack. When the function ends, the stack shrinks back to the space it was before the function call, effectively deallocating the memory for the local variables.

One drawback of the stack dynamic technique is that it does not directly support linked structures. Another issue is that we cannot change the amount of memory allocated for a variable after its first allocation. In most cases, the exact amount of memory that is allocated for a variable is determined at compile time; the one exception we saw was the variable length arrays discussed in section 8.11. In this case, the amount of memory to be allocated is not determined at compile time, but once it is allocated, we cannot make the array larger using the same variable. This makes it impossible to make a stack-dynamic-based data structure similar to Python's built-in list that can grow in size as needed.

As you might have figured out by now, C++ must support another mechanism for allocating memory for variables since the Python interpreter is written in C. C++'s other technique is known as *explicit heap dynamic*. Like Python, a section of memory known as the *dynamic memory heap* (or just *heap*) is used for these allocations and deallocations. However, as the term explicit heap dynamic implies, in C++ your program code must include instructions that directly allocate and deallocate the memory. C++ uses pointer variables to support this dynamic memory allocation and deallocation. With C++ pointers we can write code that allows us to determine and change the amount of memory allocated at run-time (rather than setting the amount at compile time). We can write data structures that can grow in size as needed and write linked structures using C++ pointers. This chapter will discuss how to use C++ pointers, how they are similar to Python references, and how to write C++ classes that use dynamic memory. We will learn how to write linked structures in C++ in Chapter 11.

10.2 C++ Pointers

In C++, a pointer variable stores a memory address. C++ requires that a pointer be defined with a specific type. The type indicates how the data at the memory address should be interpreted. Remember that internally, the computer's memory stores 1s and 0s and the type of a C++ variable tells the compiler how the code it generates should interpret those bits. Since pointer variables store an address, all pointer variables require the same amount of space (four bytes on 32-bit systems). This should remind you of Python references. A C++ pointer is a concept similar to a Python reference. The difference is that with a C++ pointer, you have access to both the address and the data that the pointer points to (i.e., the data at that address), while a Python reference gives you access only to the data that the reference points to.

C++ pointers are declared using the asterisk (`*`) as a prefix to the variable name. This indicates the variable will hold the address of a memory location where a data value of the specified type is stored. A common mistake is to forget the asterisk before each variable name when you want to declare multiple pointers in one definition statement. In the following example, `b` and `c` are declared as pointers to an `int` and `d` is declared as an `int`. The second line is also legal, although we recommend you do not use this style. Placing the asterisk immediately after the word `int` makes it appear that all variables in that statement are to be pointers to an `int`, but only `e` is a pointer and `f` is an `int`. This example allocates 20 bytes since both `int` types and pointer types require four bytes.

```
int *b, *c, d; // b and c are pointers to an int, d is an int
int* e, f; // only e is a pointer to an int, f is an int
```

The next question you should be asking yourself is how do we store an address in a pointer variable. We have no idea which memory addresses our program is allowed to use so we have to request a valid address. One way is to use the address of an existing variable. The following example demonstrates this and also shows us how to access the data that a pointer variable points to.

```
// p1.cpp
#include <iostream>
using namespace std;

```

```
int main()
{
  int *b, *c, x, y;
  x = 3;
  y = 5;
  b = &x;
  c = &y;
  *b = 4;
  *c = *b + *c;
  cout << x << " " << y << " " << *b << " " << *c << " ";
  c = b;
  *c = 2;
  cout << x << " " << y << " " << *b << " " << *c << endl;
  return 0;
}
```

The unary ampersand operator computes the address of its operand. Thus, the statement `b = &x` causes the program to store the memory address of `x` in the memory for the variable `b`. The following table indicates that the computer used memory addresses 1000 through 1015 to store our variables and shows the value of each variable after the statement `c = &y` is executed. The computer does not necessarily use the address starting at 1000, but we commonly use that address in our examples in this book.

Memory address	**Variable name**	**Data value**
1000	`b`	1008
1004	`c`	1012
1008	`x`	3
1012	`y`	5

The unary asterisk operator is used to *dereference* a pointer. Dereferencing a pointer means to access the data at the address the pointer holds. The statement `*b = 4` causes the program to store the data value 4 at memory address 1008 (since `b` currently holds 1008). Based on this knowledge, see if you can determine the output of the sample program before reading the next paragraph.

The statement `*c = *b + *c` determines the integer values at memory address 1008 (the address `b` points to) and memory address 1012 (the address `c` points to) and adds the `4` and `5` together. The result, `9`, is stored at memory address 1012 (the address that `c` points to). The statement `c = b` copies the data value for `b`, which is the address 1008, to the memory for `c` (i.e., 1008 is now stored at memory location 1004). You should note that assigning pointer variables is essentially the same as assigning two names in Python; both `b` and `c` now refer to the same data. Based on the information in the preceding paragraphs, you should be able to determine that the output of the program is `4 9 4 9 2 9 2 2`. After the statement `*c = 2`, the memory representation is

Memory address	Variable name	Data value
1000	b	1008
1004	c	1008
1008	x	2
1012	y	9

You may have already realized this, but another important concept to understand is that a pointer to an `int` and an `int` are not the same type. Using the variable declarations in the previous example, the statements `b = x` and `x = b` are not legal. The variable `b` is a pointer so it must be assigned an address whereas `x` is an `int` so it must be assigned an integer. This is more obvious if we declare the pointer variables with another type such as `double` since on 32-bit systems they do not use the same amount of storage. No matter what the type, a pointer to that type and the actual type are not compatible data types.

We will now write a more practical example demonstrating the address and dereferencing operators. The C programming language does not support pass by reference as C++ does, so the only way to effectively change the actual parameters using the C language is to use pointers. The necessary technique is to pass the address of the actual parameter and then have the function or method dereference the pointer so it changes the value at the address corresponding to the formal parameter. You can do this in C++ also, but programmers typically use pass by reference to accomplish this. The following example shows a `swap` function that swaps two integer variables.

```
// swap.cpp
#include <iostream>
using namespace std;

void swap(int *b, int *c)
{
  int temp = *b;
  *b = *c;
  *c = temp;
}
```

```
int main()
{
  int x = 3, y = 5;
  swap(&x, &y);
  cout << x << " " << y << endl;
  return 0;
}
```

The formal parameters `b` and `c` are given the values of the addresses of `x` and `y` respectively. Thus, the assignment statement `*b = *c` is equivalent to writing `x = y` in the `main` function. You should note the similarity between this and pass by reference. What happens if we add the line `b = &temp` to the end of the `swap` function; does it change `x`? The statement would have no effect on `x`. The variable `b` changes to hold the address of `temp`, but this does not change `x` or the value at the memory address corresponding to `x`.

In our examples so far, we used the unary ampersand operator to assign a valid address to a pointer variable. The other way to set a pointer to a valid address is the `new` statement. The C++ `new` statement is used to allocate dynamic memory from the heap and it returns the starting address of the memory that was allocated. When you use the `new` statement, you must indicate the data type for the object that you want to allocate; the specified data type is used to determine how much memory to allocate. When you explicitly allocate memory in C++, you must also deallocate the memory when it is no longer needed. The `delete` statement is used to deallocate memory that was dynamically allocated. The following example shows the explicit heap dynamic version of our Python and C++ program written in section 10.1.

```
// p2.cpp
#include <iostream>
using namespace std;

int main()
{
  int *x, *y, *z;
  x = new int;
  *x = 3;
  y = new int;
  *y = 4;
  z = x;
  x = y;
  cout << *x << " " << *y << " " << *z << endl;
  delete z;
  delete y;
  return 0;
}
```

The pointer variables `x`, `y`, and `z` are stack dynamic variables and the 12 bytes required for them are automatically allocated when the function begins and deallocated when the function ends. In the following table, we have used the memory locations 1000–1011 for them. The `new` statement allocates memory from the dynamic memory heap that we have started at memory location 2000. Notice that we have two `new` statements so we must have two `delete` statements. We did not use the same variable names with the `new` and `delete` statements, but the memory allocated by the `x = new int` statement is deallocated by the `delete z` statement since `z` holds the address that was allocated by that `new` statement. The `delete y` statement deallocates the memory allocated by the `y = new int` statement. We could have used `delete x` instead of `delete y` since the statement `x = y` causes both `x` and `y` to hold the same address. The key point to remember is that each `new` statement that is executed must eventually have a corresponding `delete` statement that is executed to deallocate the memory that the `new` statement allocated. If you forget a `delete` statement, your program will have a *memory leak*. Even though a program with a memory leak may not crash, the code is not considered correct.

Memory address	**Variable name**	**Data value**
1000	`x`	2000 then 2004
1004	`y`	2004
1008	`z`	2000
2000		3
2004		4

Normally you would write this program as we did in section 10.1 since that is more efficient. This pointer version requires more memory, and dereferencing a pointer requires the computer to access two memory locations (`cout << *x` requires accessing memory location 1000 followed by memory location 2004). The C++ version in this section is similar to the Python version as far as how the memory is allocated. Compare the table for this version to the memory picture in Figure 10.1. This demonstrates how Python references and C++ pointers are essentially the same concept with different syntaxes.

Since Python only uses references, it does not need the extra syntax that C++ pointers do for dereferencing a pointer. When you assign one C++ pointer variable to another, the result is they both point to the same object or value. Using pointers, we can implement the same `Rational` example we did earlier in the chapter so that the C++ version allocates memory similarly to the Python version.

One issue when using pointers to access members of a class instance is that the dot operator (the period) has a higher precedence than the asterisk (the unary *) for dereferencing a pointer. This means that if we have a `Rational` instance `r`, we cannot write `*r1.set(2, 3)`; we need to write `(*r1).set(2, 3)`. C++ provides an additional operator so we can deference a pointer and access a member without the parentheses; the notation for this is `->` (the minus sign followed by a greater-than sign) so `(*r1).set(2, 3)` can be written as `r1->set(2, 3)`. The form using `->` is more commonly used than the parentheses version.

The C++ code using C++ pointers that corresponds to the same Python `Rational` example earlier in the chapter is the following

```
Rational *r1, *r2; // constructors not called

r1 = new Rational; // constructor is called
r1->set(2, 3);
r2 = r1;
r1->set(1, 3);

cout << *r1 << endl;
cout << *r2 << endl;
delete r1;
```

This example outputs `1/3` for `r1` and `1/3` for `r2` since `r1` and `r2` are pointers to the same memory locations. The memory table for this code fragment is:

Memory address	**Variable name**	**Data value**
1000	`r1`	? then 2000
1004	`r2`	? then 2000
2000		2 then 1
2004		3

The declarations of `r1` and `r2` result in four bytes being allocated for each one since pointers require four bytes. The `Rational` constructor is not called when you declare a pointer since we are creating a pointer, not a `Rational` object. The statement `r1 = new Rational` results in eight bytes being allocated since the two integer instance variables `num_` and `den_` require a total of eight bytes. The `r1 = new Rational` statement also causes 2000 to be stored in the memory location for variable `r1`. The constructor is called by `r1 = new Rational` since it creates a `Rational` object. The `r1->set(2, 3)` statement results in `2` being stored at memory location 2000 and `3` being stored at memory location 2004.

The statement `r2 = r1` results in 2000 being stored in memory location 1004 since the value of `r1` is 2000. We now effectively have the same memory structure as our Python example with both `r1` and `r2` referring to the same `Rational` object. When we execute `r1->set(1, 2)` statement we are not changing `r1` but are changing the object stored at the memory location that `r1` points to. Since `r2` points to the same object as `r1` we get the same results as we do in Python. When the function containing our C++ code fragment ends, the memory locations for the declared variables (1000–1007) is automatically deallocated as we discussed earlier, but we need the `delete r1` statement to deallocate the memory at locations 2000–2007 which we explicitly allocated with the `new Rational` statement. We could have written `delete r2` instead since both pointers refer to the same locations, but we cannot write `delete r1; delete r2` since each `new` statement must have one and only one corresponding `delete` statement. Trying to delete the same memory locations a second time may corrupt the dynamic memory heap, resulting in a crash.

Using pointers with dynamic memory in C++ gives you the flexibility of Python references, but because you are in charge of explicitly handling the allocation and deallocation, it is much more difficult to get correct than Python versions of the same code. If you are not careful when using dynamic memory, your program can produce different results each time you run it or may crash. We will discuss these issues for explicit heap dynamic memory throughout this chapter.

10.3 Dynamic Arrays

The built-in array data structure with a fixed size was discussed in section 8.11. In many cases, we do not know the size of the array at compile time or we want to change the size of the array as the program is running, so we need a mechanism for allocating an array of a specified size at run-time. As we saw in the previous section, C++ pointers can be used to *dynamically allocate* memory. This means that the memory is allocated as the program is running and the amount of memory allocated may be determined at run-time instead of being set at compile-time. The following code fragment demonstrates dynamic memory allocation and deallocation for arrays:

```
int i, n;
double *d;

cout << "Enter array size: ";
cin >> n;
```

```
d = new double[n];
for (i=0; i<n; ++i) {
  cout << "Enter number " << i << ": ";
  cin >> d[i];
}
delete [] d;
```

The example allows the user to specify the array size at run-time. The `new` command allocates the specified amount of memory and returns the starting address of the allocated memory. When the brackets (`[]`) are used after the data type in the `new` statement, the amount of memory necessary to store the number of items specified inside the brackets is allocated and the starting address is returned. In this case, n*8 consecutive bytes would be allocated on machines that use eight bytes to store a double value. The expression inside the brackets indicates how many values of the type double to allocate; an array of size `n` was allocated so the valid index values are 0 through n-1. After the dynamic memory has been allocated, it can be accessed using the array bracket notation. The same index array calculations discussed in section 8.11 can be used since the pointer variable holds the starting address of a contiguous section of memory.

Whenever you allocate memory dynamically, you must also deallocate the memory with a statement that executes later in your program. Since we allocated an array, we must tell the `delete` statement to deallocate an array instead of the memory that holds a single value. The square brackets are used with both the `new` statement and the `delete` statement when allocating and deallocating arrays. You do not indicate the size of the array when deallocating a dynamic array; the C++ run-time environment knows how much memory to deallocate. Repeatedly allocating memory and forgetting to deallocate memory in a C++ program will eventually result in your program using up a large percentage of the computer's memory, causing the computer to slow as it uses the hard disk for extra memory. This is why it is important to deallocate memory when it is no longer needed.

The main reason for using dynamic arrays is that you do not need to know the size of the array at compile time. In many cases, you still may not know the size needed when the array is first allocated. The Python built-in list allows you to append as many items as you want so there is no need to determine how much memory to allocate the first time you allocate memory; it would be impossible to anticipate how much memory to allocate ahead of time since different uses of the list will require different sizes. Once the array fills up, we may need to make the array larger. Because the memory immediately following the dynamic array may already be in use (remember that array elements must be in consecutive memory locations), we cannot make the array larger. The solution is to allocate a new larger array, copy the values from the original array to the new array, and then delete the original array. The following code fragment demonstrates this:

```
int *data, *temp;
int i;

// create original array
data = new int[5];
for (i=0; i<5; ++i) {
  data[i] = i;
}
// create new larger array
temp = new int[10];
// copy from original array to larger array
for (i=0; i<5; ++i) {
  temp[i] = data[i];
}
// deallocate original array
delete [] data;
// make data point to new larger array
data = temp;
// now we can access positions 0-9
for (i=5; i<10; ++i) {
  data[i] = i;
}
// deallocate last allocation
delete [] data;
```

The memory table for this code after the first `new` statement and `for` loop are executed is below. We will assume the memory addresses used for the local variables start at memory location 1000 and that the dynamically allocated memory is the block of memory from 2000 through 2019 (four bytes for each of the five integers).

Memory address	**Variable name**	**Data value**
1000	`data`	? then 2000
1004	`temp`	?
1008	`i`	5
2000		0
2004		1
2008		2
2012		3
2016		4

After the memory is allocated for the `temp` pointer, the values are copied from the original array, and the values 5 through 9 are stored in the larger array, the memory table is the following assuming the memory starting at location 3000 is used for the `temp` pointer.

Memory address	**Variable name**	**Data value**
1000	`data`	2000
1004	`temp`	3000
1008	`i`	10
2000		0
2004		1
2008		2
2012		3
2016		4
3000		0
3004		1
3008		2
3012		3
3016		4
3020		5
3024		6
3028		7
3032		8
3036		9

After the first `delete [] data` statement, the memory at locations 2000–2019 is deallocated and returned to the dynamic memory heap so it can be used again. The statement `data = temp` stores 3000 at memory location 1000 (i.e., `data` now points to the second larger allocated array). At this point, both `data` and `temp` point to the same dynamically allocated array. This is the same concept as having two references to the same data in Python. After that assignment statement, the next loop fills in the values 5 through 9 in memory locations 3020 through 3039. The final `delete [] data` statement then deallocates the memory locations 3000–3039 so they can be used again.

Figure 10.2 shows a pictorial representation of this. The top part of the figure shows the representation after we have created the new larger array and copied the values from the first array. The middle part of the figure shows the state after the first `delete data` statement. The bottom part of the figure shows the state just before the final `delete [] data` statement.

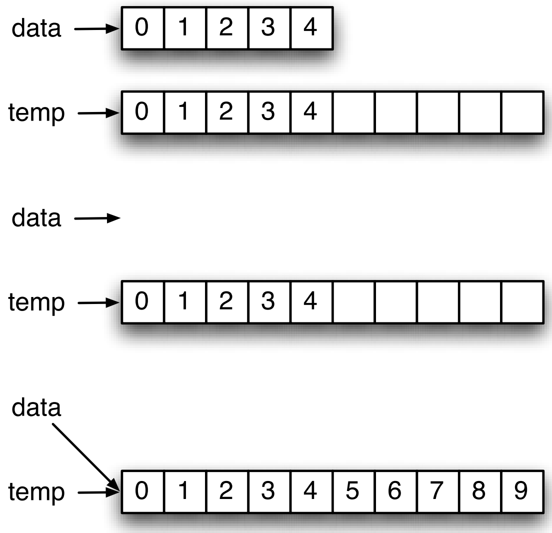

Figure 10.2: Pictorial representation of resizing a dynamic array

If we then fill up this new array and need a larger array, we allocate a larger array, copy the values from the previously allocated array, and then delete the previous array. Each resizing operation results in the previous array being deleted so that we do not have a memory leak if we perform this resizing operation multiple times. In our example, `data` points to the last array that was allocated (once we execute the `data = temp` statement). This pattern of allocating a new section of dynamic memory using a different pointer variable, copying the values from the old section to the new section, deallocating the old section, and setting the original pointer variable to the new section is a common pattern in C++ dynamic memory, so make certain you fully understand how it works and why the order of the steps is important. In the next section, we will examine this pattern using a class.

10.4 Dynamic Memory Classes

When you write a class that dynamically allocates memory for pointer instance variables, you need to make certain that the memory is properly deallocated. There are three additional C++ methods that dynamic memory classes must use to properly allocate and deallocate memory. These three methods are the destructor, copy constructor, and assignment operator (`operator=`). If your class does not use dynamic memory, you do not need to write any of these methods. Classes that use dynamic memory must write a destructor that deallocates the memory. The other two methods may be implemented or declared in the `private` section but not implemented. Declaring them in the `private` section but not implementing them prevents them from being called. We will discuss the details of when these methods are called and what they must do in this section. Implementing them correctly will prevent your class from having memory leaks or other memory errors.

10.4.1 Destructor

As discussed in the previous sections, in C++ you must explicitly deallocate any memory that you explicitly allocate with the `new` command. C++ classes have a special method known as a *destructor* that is used for deallocating memory. The destructor method has the same name as the class, with a tilde (~) in front of it. Just as constructors do not have a return type, the destructor also does not have a return type. The purpose of the destructor is to deallocate any dynamic memory the class has allocated that has not yet been deallocated. You never directly call the destructor using the name of the method; it is called automatically when an instance of the class goes out of scope or when you use the `delete` operator on a pointer to an instance of the class. If your class uses dynamic memory and does not have a destructor, your code will in most cases have a memory leak.

We will start with a simple dynamic array class that we will extend throughout this chapter to demonstrate how to correctly write dynamic memory classes. In this first version of the class, we will write all the methods inline in the header file. We have added a few output statements so you can see that the constructor and destructor are called. This `List` class uses three instance variables. The instance variable `data_` is used to hold the starting address of the dynamic array containing the list's values. The `size_` instance variable indicates how many items are currently in the list. The `capacity_` instance variable indicates how large the dynamic array is (i.e., how many items the list can hold before the dynamic array needs to be resized).

```
// List1.h
#ifndef __LIST_H__
#define __LIST_H__

#include <iostream>

class List {

public:
  List(int capacity=10);
  ~List() { delete [] data_; std::cout << "destructor\n";}

private:
  int size_;
  int capacity_;
  int *data_;
};

inline List::List(int capacity)
{
  std::cout << "constructor\n";
  data_ = new int[capacity];
  size_ = 0;
  capacity_ = capacity;
}

#endif __LIST_H__
```

We did not put the `using namespace std` statement in the header file since any file that included the header file would have this statement. Also note that we put the default value for the parameter `size` only in the declaration of the `List` constructor and not in the implementation of the constructor. Here is a simple program that uses our class:

```
// test_List1.cpp
#include <iostream>
using namespace std;

#include "List1.h"

int main()
{
  List b;
  return 0;
}
```

When this program is compiled and executed, it outputs

```
constructor
destructor
```

The declaration `List b` causes the constructor to be called and it allocates dynamic memory. At the end of the `main` function, the variable `b` goes out of scope so the destructor method is automatically called and it deallocates the dynamic memory. This is why the program outputs the two lines. The following program also causes the same output.

```
// test_Listp.cpp
#include <iostream>
using namespace std;

#include "List1.h"

int main()
{
  List *b; // constructor is not called here

  b = new List(20); // constructor is called here
  delete b; // destructor is called here
}
```

The comments indicate when the constructor and destructor are called. Remember the declaration `List *b` causes four bytes that can store an address to be allocated. The `new` statement causes a `List` object to be created by calling the constructor with the specified size. The `delete` statement causes the `List`'s destructor to be called and the dynamic memory the constructor allocated is deallocated. When the variable `b` goes out of scope at the end of the `main` function, the four bytes for the pointer are automatically deallocated just as the memory for any variables are when they go out of scope. This is the reason you only need to write a destructor when your class allocates dynamic memory.

10.4.2 Copy Constructor

As the name implies, the purpose of a *copy constructor* is to create a new object by copying an existing object. In C++, the copy constructor for a class is called when you pass an instance of a class by value to a function or method. Remember that pass by value requires that a separate copy of the actual parameter be created. You can also call a copy constructor directly when declaring a variable as we will demonstrate later in this section.

Unless you write a copy constructor for a class, the C++ compiler generates a default copy constructor for you. The default copy constructor it creates effectively assigns each instance variable of the existing object to the corresponding instance variable in the newly created object. For a class that does not use dynamic memory and pointers, this is exactly what you want. Consider our `Rational` class we have been discussing. To create an exact copy of a `Rational` object, we want to assign its numerator and denominator instance variables and this is exactly what the default copy constructor does.

For classes that use dynamic memory, the default copy constructor will create a shallow copy of the dynamically allocated data; the pointer variable in both instances will refer to the same section of dynamically allocated memory. This will cause problems. When the destructor for one of the objects is called, it will deallocate the dynamic memory that is shared by both objects. The other object can no longer legally access that data and when the destructor is called for it, it will attempt to deallocate the same memory a second time. The second deallocation is illegal and will lead to memory corruption errors that can cause your program to give incorrect results or crash. As we discussed earlier, each memory section that is dynamically allocated must be deallocated exactly once.

We will continue extending our dynamic array example by adding the copy constructor to it. Since it is a constructor, it has the same name as the class and since it is to copy an instance of the class, we must pass that instance as a parameter. Remember that the copy constructor is called when we pass an instance of the class by value. If the copy constructor parameter was passed by value, it would need to call itself to make a copy, leading to an infinite number of calls. The copy constructor parameter must be passed by reference to avoid this. Remember that to pass an object by reference, all that needs to be done is pass the address of the object. We showed the equivalence of this with the `swap` function we wrote in section 10.2. Here is the updated header file for the dynamic array class with a copy constructor added.

```
// List2.h
#ifndef __LIST_H__
#define __LIST_H__

#include <iostream>

class List {

public:
  List(int capacity=10);
  List(const List &source);
  ~List() { delete [] data_; std::cout << "destructor\n";}
```

```
private:
  int size_;
  int capacity_;
  int *data_;
};

inline List::List(int capacity)
{
  std::cout << "constructor\n";
  data_ = new int[capacity];
  size_ = 0;
  capacity_ = capacity;
}

inline List::List(const List &source)
{
  int i;

  std::cout << "copy constructor\n";
  size_ = source.size_;
  capacity_ = source.capacity_;
  data_ = new int[capacity_];
  for (i=0; i<size_; ++i) {
    data_[i] = source.data_[i];
  }
}

#endif __LIST_H__
```

As the code shows, the copy constructor parameter is passed by reference and with the `const` designation so that the method does not need to call itself and does not change the existing data. You can use any name you want for the formal parameter, but one common convention is to name it `source` to indicate this is the source instance of the class that you are copying. When we refer to `size_`, that is the instance variable for the new object we are creating. When we refer to `source.size_`, that is referring to the instance variable of the object we are copying. You may be surprised that we can refer to the other object's instance variables using the code such as `size_ = source.size_` since the instance variables are private; however, we are writing a method of the class so it is allowed to access the private data of any instance of the class, not just the instance with which it is called.

For the instance variables that are not pointers, we want to assign each one of them so the newly created object has the same values for the size and capacity. We then need to allocate a new array with the same capacity and copy the elements from the source object's array into it. This will create a deep copy. Notice that we only copied up to the value of `size_` since the values past that are not relevant to the object. At this point we do not have any way of putting elements in our simplified class to cause `size_` to be more than zero, but our final version will. The following example using this class allows us to see when the copy constructor is called.

```
// test_List2.cpp
#include <iostream>
using namespace std;
#include "List2.h"

void f(List c)
{
  cout << "start f\n";
  cout << "end f\n";
}

void g(List &d)
{
  cout << "start g\n";
  cout << "end g\n";
}

int main()
{
  List b;
  f(b);
  g(b);

  List e(b);
  return 0;
}
```

The output of the program is the following. The comments in parentheses are obviously not part of the output, but explain what caused each of the methods to be called.

```
constructor (create b in main function)
copy constructor (create the copy c from b in f function)
start f
end f
destructor (destructor for c when function f completes)
start g
end g
copy constructor (create e in main function)
destructor (destructor for e or b)
destructor (destructor for e or b)
```

The first issue to note is the implicit call of the copy constructor when the function f is called. The copy constructor executes before the function begins execution to make a copy of the parameter. After the function completes, the destructor is automatically called to deallocate the memory dynamically allocated by the copy constructor. Since the function g passes the parameter by reference, the copy constructor is not called. Also, the destructor is not called for d when the function g completes; if it were, we would be deallocating the dynamic memory for the variable b in the main function. The statement List e(b) explicitly causes the copy constructor to be called to create e from the existing object b. When the main function completes, both e and b are destructed to deallocate their dynamic memory. You should not rely on the order that the destructor is called for e and b. All you need to care about is that both objects will be properly destructed when the main function completes. As this example demonstrates, if you correctly write each method, the rules for when the constructor, copy constructor, and destructor are called will correctly allocate and deallocate memory.

As we mentioned earlier, you can declare the copy constructor private and not implement it. This prevents code that uses your class from causing the copy constructor to be called; the code would not be able to pass an instance of your class by value to a function or method or explicitly call the copy constructor. Code that attempts to perform either of those actions will generate a compiler error. If your class uses a large amount of memory, you may want to do this to prevent a user of your class from making a copy of it. The following header file demonstrates this.

```
// List3.h
#ifndef __LIST_H__
#define __LIST_H__

#include <iostream>

class List {
public:
  List(int capacity=10);
  ~List() { delete [] data_; std::cout << "destructor\n";}
private:
  List(const List &source);
  int size_;
  int capacity_;
  int *data_;
};
```

```
inline List::List(int capacity)
{
  std::cout << "constructor\n";
  data_ = new int[capacity];
  size_ = 0;
  capacity_ = capacity;
}

#endif __LIST_H__
```

10.4.3 Assignment Operator

The other method you must write or declare private when using dynamic memory is the `operator=` method. This method is called when you assign an instance of your class to another instance of the class (e.g., `b = c`). This is a very similar operation to the copy constructor except that the instance on the left-hand side (`b` in the example) already exists so it already has dynamic memory allocated for it. When the copy constructor is called, the object has not yet been allocated, but for the assignment operator, the constructor was previously called with the object so it likely has dynamic memory already allocated for it.

Similarly to the copy constructor, the compiler will write a default assignment operator for your class if you do not write one. It will do what you expect and assign each instance variable individually. If your class does not use dynamic memory, this is exactly what you want. For the same reasons discussed for the copy constructor, you do not want this for classes that use dynamic memory; it will result in two instances of the object sharing the same dynamically allocated memory. The following header file demonstrates the three methods you need to write. We have written these examples with inline methods in the header file to keep the examples shorter, but we could have written them with a separate implementation file. We have removed the output statements now that we know when each method is called.

```
// List4.h
#ifndef __LIST_H__
#define __LIST_H__

class List {

public:
  List(int capacity=10);
  List(const List &source);
  ~List() { delete [] data_; }
  void operator=(const List &source);
```

```
private:
  int size_;
  int capacity_;
  int *data_;
};

inline List::List(int capacity)
{
  data_ = new int[capacity];
  size_ = 0;
  capacity_ = capacity;
}

inline List::List(const List &source)
{
  int i;

  size_ = source.size_;
  capacity_ = source.capacity_;
  data_ = new int[capacity_];
  for (i=0; i<size_; ++i) {
    data_[i] = source.data_[i];
  }
}

inline void List::operator=(const List &source)
{
  int i;

  if (this != &source) {
    delete [] data_;
    size_ = source.size_;
    capacity_ = source.capacity_;
    data_ = new int[capacity_];
    for (i=0; i<size_; ++i) {
      data_[i] = source.data_[i];
    }
  }
}

#endif __LIST_H__
```

Since the object has already been created, the assignment operator is a little more complicated than the copy constructor. We must properly deallocate memory that has already been allocated and ensure that the class is not accidently assigning the object to itself or we will deallocate the only copy of the data. In C++ classes, the identifier `this` is an implicit pointer to the object with which the method is explicitly called. For example, if we have two `List` objects `b` and `c` and write `b = c`, this is equivalent to writing `b.operator=(c)`; review section 9.4 if you need a refresher on operator overloading. The assignment operator must be written as a member of the class; it cannot be written as a standalone function as many of the other operators can. For the assignment statement `b = c`, the `this` pointer will hold the address of `b`. The `this` pointer is equivalent to the explicit `self` reference that all Python methods have. We could use the `this` pointer to explicitly refer to all instance variables and methods such as `this->size_` instead of just `size_` if we wanted to, but most C++ programmers do not use this style.

The `if` statement in the method checks if the method on the left-hand side of the assignment statement (`b` in our example) is the object at the same address as the object on the right-hand side of the assignment statement (`c` in our example). If they are the same object, we do not want to do anything. Deleting the dynamic memory would delete the one copy of dynamic memory. You may have noticed that the copy constructor and assignment operator share most of the code; because of this, it is common to write the shared code in a private method that both the copy constructor and assignment operator call. We will demonstrate this in our final version of the dynamic array class later in this section.

You might be wondering how a programmer could end up assigning an object to itself. Certainly, no programmer would write `b = b;` in their code and it would be possible to write a compiler to catch this mistake. You need to remember that since we can use pointers, we can end up with two pointers with different names referring to the same object. The following example is still contrived, but you can imagine a function that would return a pointer to a `List` object and the programmer would not have any idea what other `List` pointer variables also point to it.

```
#include "List3.h"

int main()
{
  List *b, *c, d;
  b = &d;
  c = b;

  *b = *c; // causes operator= to be called
  return 0;
}
```

In the example, both `b` and `c` refer to the `List` object that is the variable `d` so the statement `*b = *c` causes the `List::operator=` method to be called. Notice that the statement `b = c` does not call the `List::operator=` method to be called.

The variables `b` and `c` are pointers so this is the assignment of two pointers, causing them both to store the same address.

10.4.4 A Complete Dynamic Array Class

We will now write a realistic version of the `List` class, adding a few more new concepts to the ones we discussed earlier. Without the use of dynamic memory, we could not write a `List` class in C++ that could grow beyond the initial size of the array. The following example shows all the methods necessary to correctly implement a realistic use of dynamic memory (a copy constructor, assignment operator, and destructor). In the following example we use the data type `size_t`, which is a synonym for an `unsigned int` (i.e., a non-negative integer), for the instance variables and parameters that specify a position in the array since an array cannot have a negative size. There are some potential pitfalls with using an `unsigned int` that we will discuss later in the section.

```
// List.h
#ifndef _LIST_H_
#define _LIST_H_

#include <cstdlib>
class List {
public:
  List(size_t capacity=10); // constructor - allocates dynamic array
  List(const List &a); // copy constructor
  ~List(); // destructor

  int& operator[](size_t pos); // bracket operator
  List& operator=(const List &a); // assignment operator
  List& operator+=(const List &a); // += operator
  void append(int item);
  size_t size() const { return size_; }
private:
  void copy(const List &a);
  void resize(size_t new_size); // allocate new larger array
  int *data_; // dynamic array
  size_t size_; // size of dynamic array
  size_t capacity_; // capacity of dynamic array
};

inline int& List::operator[](size_t pos)
{
  return data_[pos];
}
#endif // _LIST_H_
```

The bracket operator (`operator[]`) provides the same functionality as the Python `__getitem__` and `__setitem__` methods. It is declared inline after the class definition and demonstrates a reference return type. The ampersand after the type name indicates that a reference to an integer is returned, meaning it effectively returns the address of the position in the array. This allows the operator to be used on the left-hand side of an assignment statement as `b[0] = 5` where `b` is an instance of our `List` class. It can also be used on the right-hand side of an assignment statement or as part of an expression just as a non-reference return type can. Without the reference return type, the operator could only be used on the right-hand side of an assignment statement (corresponding to only the Python `__getitem__` method). Returning a reference only makes sense if it is a reference to an instance variable or dynamically allocated memory. We will discuss this later in the chapter.

The `List` class provides an array of integers whose initial size is specified when the constructor is called. The constructor allocates a dynamic array with the specified capacity and initializes the `size_` instance variable to indicate the list is empty. If we did not allocate the memory in the constructor, but instead deferred it to another method (such as the first time the `append` method is called), we would initialize the pointer variables to `NULL`. The `NULL` constant is defined in the `cstdlib` header file. The value `NULL` is defined to be zero which is never a valid address for memory that has been dynamically allocated. The use of `NULL` in C++ to indicate an invalid pointer is similar to the use of `None` in Python to indicate a reference that is not initialized to an object of a specific type.

The class makes use of a private method named `copy` to implement the code that is needed in both the copy constructor and assignment operator. The assignment operator needs extra code to deallocate the existing dynamic array before allocating a new dynamic array of the appropriate size and copying the data. Remember that a copy constructor is creating a new object, so no memory has been previously allocated for the object when the copy constructor is called. However, the variable on the left-hand side of an assignment statement has already had its constructor called and memory allocated, so that needs to be deallocated. The code for the destructor follows the copy constructor. The destructor simply deallocates the dynamic array and is called automatically when a non-pointer instance of `List` goes out of scope. Note that it does not deallocate the non-pointer instance variables since the memory for those is automatically deallocated. In this example, we are using a separate implementation file unlike the earlier simplified examples in which the entire class was written in the header file.

```
// List.cpp
#include "List.h"

List::List(size_t capacity)
{
  data_ = new int[capacity];
  capacity_ = capacity;
  size_ = 0;
}

List::List(const List &list)
{
  copy(list);
}

List::~List()
{
  delete [] data_;
}
```

We have written the `operator=` method slightly differently so that we can use it in a chained assignment statement. The method returns a reference to a `List` object. By returning `*this`, we are returning the `List` object that we just assigned. This allows us to write the chained form of the assignment statement (e.g., `b = c = d`. Remember that the assignment operator is right to left so it is equivalent to `c = d; b = c`. By returning a reference to the left-hand parameter, the result of `c = d` is the object `c` that we then use as the right-hand parameter when assigning `b`.

The `copy` method that is used by both the `operator=` and the copy constructor allocates an array of the same size as the `List` object that is passed to it and copies all the data from the parameter object's array into the newly allocated array. We have also added the `operator+=` method so we can demonstrate another potential pitfall.

```
void List::copy(const List &list)
{
  size_t i;
  size_ = list.size_;
  capacity_ = list.capacity_;
  data_ = new int[list.capacity_];
  for (i=0; i<list.capacity_; ++i) {
    data_[i] = list.data_[i];
  }
}
```

```
List& List::operator=(const List &list)
{
  if (&list != this) {
    // deallocate existing dynamic array
    delete [] data_;
    // copy the data
    copy(list);
  }
  return *this;
}
List& List::operator+=(const List &list)
{
  size_t i;
  size_t pos = size_;
  if ((size_ + list.size_) > capacity_) {
    resize(size_ + list.size_);
  }

  for (i=0; i<list.size_; ++i) {
    data_[pos++] = list.data_[i];
  }
  size_ += list.size_;
  return *this;
}
```

The `operator+=` appears straightforward, but if you are not careful, subtle errors can be introduced. If we replace the last few lines with the following code so that it increments the `size_` variable as it adds the items onto the array, it will work fine in most cases.

```
  // this version is incorrect
  for (i=0; i<list.size_; ++i) {
    data_[size_++] = list.data_[i];
  }
```

What happens if we have a `List` instance `b` and execute `b += b`? In this case, `size_` and `list.size_` are two names for the same memory location (i.e., they are both bound to the same address). Since we are incrementing `size_` each time through the loop, the `for` loop will never end because `i` will always be less than `list.size_`. These types of subtle errors can be extremely difficult to track down so always consider these special cases when writing your own code and test for them.

The `append` method is straightforward except that we may need to allocate a larger array if we have already filled the existing array. We have written a separate `resize` method that the `append` method calls when necessary to perform the steps of allocating a new larger array, copying the data to it, updating the pointer, and then deallocating the old smaller array, as we discussed in section 10.3.

```
void List::append(int item)
{
  if (size_ == capacity_) {
    resize(2 * capacity_);
  }
  data_[size_++] = item;
}

// should this method have a precondition? see end of chapter exercises
void List::resize(size_t new_size)
{
  int *temp;
  size_t i;

  capacity_ = new_size;
  temp = new int[capacity_];
  for (i=0; i<size_; ++i) {
    temp[i] = data_[i];
  }
  delete [] data_;
  data_ = temp;
}
```

We leave it as an exercise to add the other methods in the built-in Python list's API to this C++ dynamic memory list. As we mentioned earlier, you do need to be careful when using `unsigned int` or the equivalent `size_t` data type. As we listed in Figure 8.4, the range of the `int` type on 32-bit systems is usually from about negative two billion to about positive two billion while the `unsigned int` type ranges from zero to about four billion. With the `unsigned int` data type, there is no bit representation that corresponds to a negative number. So the question is, what happens when an operation would result in a negative number?

```
// unsigned.cpp
#include <iostream>
using namespace std;
int main()
{
  unsigned int x = 0;
  x--;
  cout << x << endl;
  return 0;
}
```

The output of this operation in a program compiled for 32-bit systems is 4294967295. This is the largest possible integer that can be represented with 32 bits (the bit representation is 32 1s). We have *overflowed* the bit representation. This is like going beyond the number of digits in a car odometer. Think about what would happen if you were able to run a car odometer backwards past zero; you would get the largest value the odometer can hold. This is essentially the same thing that happens when you overflow integer values on the computer. C++ does not automatically indicate when overflow occurs. There are ways to detect it, but we will not cover these details in this book. When writing your code, you must ensure that you do not accidently overflow the range of values the data type you are using can store or you will get unexpected or incorrect results. The next code fragment demonstrates an error caused by overflow.

```
unsigned int i;
unsigned int pos=0;
for (i=5; i>=pos; --i) {
  cout << i << endl;
}
```

If you create a program with this loop and run it, you probably expect the loop to execute six times (the expression `i >= pos` should be true when `i` is five, four, three, two, one, and zero). The problem is that after setting `i` to zero, the next value for `i` will be 4294967295 and that is obviously also greater than or equal to zero so this produces an infinite loop. If `pos` were any positive value, this would not occur. It is always a good idea to test your code with these boundary conditions to ensure it works in all cases.

10.4.5 Reference Return Types

As we mentioned earlier, you should not return a reference to a local variable. The reason for this is that a reference effectively returns the memory location where the variable is stored, not a value. The problem with this is local variables in a function are automatically deallocated when the function ends. Using formal terminology, the *lifetime* of local variables is the time while the function is being executed. Once the function ends, the memory locations used for local variables are reclaimed and are no longer *bound* to those local variables. You can only return by reference a variable whose lifetime does not end when the function or method ends. The following example shows an example that returns a reference to a local variable and is incorrect; most compilers will generate a warning.

```
// this is incorrect
int& f()
{
  int x;
  return x;
}

int main()
{
  f() = 5;
}
```

In our section on the `List` class, we discussed that since the `operator[]` returned a reference, we can write `b[0] = 5` where `b` is an instance of our `List` class. On the left-hand side of the assignment statement, we are calling the operator and it returns a reference to the memory location. That memory location is then used to store the value `5`. In the previous example, the statement `f() = 5` is attempting to do the same thing; the memory location for the variable `x` returned by the function `f` is being used to store the value `5`. The problem is that the memory location is no longer being used for the local variable `x` after the function ends.

As our `List` code shows, it is correct to return a reference to an instance variable of a class instance. An object's instance variables have the same lifetime as the instance of the class. The statement `b[0] = 5` where `b` is an instance of our `List` class is equivalent to `b.data_[0] = 5`, but this is not allowed since `data_` is a private member of the class. The bracket operator is a public method and returns a reference to the private data, allowing us to legally access the private data directly. In many cases this is bad programming style, but for a class that encapsulates a dynamic array, one could argue it makes sense.

A precondition for the `operator[]` method is that the specified index is between 0 and `size_` - 1. To prevent a user of the class from crashing the program by passing an index outside of the list size, we could check that the specified index is between 0 and `size_` - 1 before attempting to access that position in the dynamic array. This extra overhead is not necessary if the code that uses the class always meets the precondition. A common technique is to include code that checks the precondition while testing and debugging your program, but once you are convinced your program is correct, you can remove the code that checks the precondition to get a small performance boost.

10.5 Dynamic Memory Errors

Using pointers in C++ gives your programs more flexibility and capabilities, but is also more error prone. Pointers to data objects also require extra memory since you need to store both the pointer and the data while the C++ default stack dynamic variables only need memory to store the data. Dynamic memory errors are the source of a large percentage of errors in most large programs. Because of these reasons, you should use dynamic memory only when you need the extra capabilities it gives you.

Dynamic memory errors are often difficult to track down and correct since sometimes your program may run fine, other times it may run but give incorrect results, and other times it may crash. We suggest you learn how to use the debugger that your programming environment supports to help you track down these memory errors. You can try to find the errors by putting output statements throughout your code, but learning how to use your debugger will save you a lot of time and frustration in the long run. Adding to the difficulty of tracking down these errors is that often the statement that causes the program to crash is not the statement that is incorrect so it is also important to proofread your dynamic memory code. In this section, we will discuss the different types of errors that can occur with dynamic memory.

10.5.1 Memory Leaks

We have already briefly mentioned one type of error known as a *memory leak*. A memory leak occurs when you allocate memory but never deallocate it. If your program repeatedly calls a function or method that leaks memory, your program will eventually require more memory than the computer has. This will lead the operating system to use the disk as extra memory. Since the disk is much slower than memory, your computer will slow down. Fortunately, when a program completes, the operating system reclaims any memory the program was using so a memory leak should not crash your program. If the operating system itself has a memory leak, it will eventually run out of memory. This is the reason some people recommend you reboot your computer occasionally.

The code examples with errors in this section are short examples that you would not normally write, but show the errors that can occur as part of larger sections of code. This first example executes two `new` statements, but executes only one `delete` statement.

```
// this code is incorrect
void f()
{
  int *x;
  x = new int;
  *x = 3;
  x = new int;
  *x = 4;
  delete x;
}
```

Figure 10.3 shows a pictorial representation of the memory leak code. The left part shows the result after the line `*x = 3` is executed; four bytes have been dynamically allocated with `x` holding the address and the value `3` is stored at that address. The middle part of the figure shows the result after the second `x = new int` statement is executed. We no longer have any way to access the dynamic memory that was originally allocated by the first `x = new int` statement. The right part of the figure shows the result after the `delete x` statement is executed; the variable `x` points to a memory location that can no longer be used and the memory location containing the `3` still exists and cannot be deallocated since we do not have a variable holding its address. This is the memory leak. To fix it, we would need another `delete x` statement before the second `x = new int` statement.

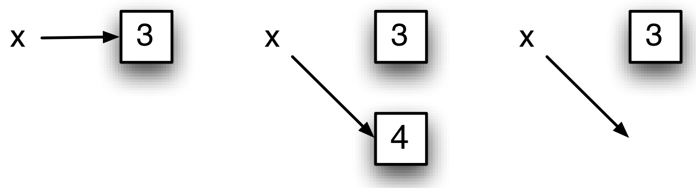

Figure 10.3: Pictorial representation of a memory leak

In many cases, the `delete` statement that deallocates the memory allocated by a `new` statement is not in the same function or method. This makes it more difficult to detect memory leaks. If you refer back to the `resize` method in our `List` class, you will notice that the `delete` statement in it is not deallocating the memory that was allocated by the `new` statement that executed earlier during the function call. The first time the `resize` method is called the `delete` statement is deallocating the memory allocated by a constructor. Each subsequent time the `resize` method is called, it is deallocating the memory allocated by the previous call to the `resize` method. This may make you think that we have a memory leak, but remember that the destructor will deallocate the memory that was allocated by the last call to the `resize` method or the memory allocated by the constructor if the `resize` method was never called.

In fact, in most cases the corresponding `new` and `delete` statements are not in the same function or method, making it difficult to be certain your code does not have any memory leaks. We will see another example of this with linked structures in Chapter 11. Proofreading and checking your code carefully is important to help prevent these errors. Some development environments provide tools to track the memory usage as your program executes so that you can watch for unexpected growth in memory usage. You may think memory leaks are not an issue to be concerned with since the operating system will reclaim any memory your program used when the program exits, but many programs run for long periods of time. A web server for a commercial site might be expected to run for months at a time without being restarted. If you do not reboot your computer regularly (letting it go into sleep or hibernate mode is not equivalent to rebooting it) and leave programs such as your email or web browser programs running all the time, you do not want these programs to have memory leaks. If these programs you leave running (or the operating system itself) have memory leaks, your computer will slow down over time until you reboot it as it starts using the disk as extra memory. Thus, it is important to get in the habit of writing code without memory leaks. The key point to remember is that each `new` statement that is executed must have a corresponding `delete` statement that is executed after your program is done using the memory allocated by the `new` statement.

10.5.2 Accessing Invalid Memory

Modern computer hardware provides checks to make certain that one program does not access memory that is used by another program. This prevents a number of problems such as one program causing another program to give incorrect data or crash. If one program could access the memory used by another program such as a web browser, it would be possible for the program to access the passwords and other sensitive information you type into a web browser. Computer hardware splits the memory into sections that are known as *pages*. On most modern computers a page is either 4KB or 8KB in size. As the amount of memory in computers continue to grow, it is likely that the page size will increase. The hardware provides protection at the page level. If a program attempts to access memory that is not in one of the pages your program is using, a hardware exception is generated and the program crashes.

Since the hardware detects errors only when a program attempts to access memory that is not one of the pages the operating system allocated for the program, a program may access memory that is one of its pages but is not a valid memory address that it should be using. In this case, your program will not crash immediately, but it can have different results each time it runs or it can crash at a later point in time.

We will start with a simple example that does not use dynamic memory, but could give unexpected results. See if you can find the error in this program before reading the paragraph after the code.

```
// this program is incorrect
#include <iostream>
using namespace std;

int main()
{
  int x[10];
  int y = 0;
  int i;

  for (i=0; i<=10; ++i) {
    x[i] = i;
  }
  cout << "y=" << y << endl;
  return 0;
}
```

Unlike Python, C++ does not do any index checking when you attempt to access an element in an array (Python does check when you attempt to access an element in a list). The problem with this example is the array can be indexed using the values `0` through `9`, but the `for` loop sets `x[10]`. Depending on how the memory is allocated for the local variables, this could result in the program outputting `10` for `y` even though we set `y` to zero. If the memory location used for the variable `y` is immediately after the memory location for the array, then `x[10]` and `y` correspond to the same memory address. If the memory for the variables is not allocated in this order then the program produces the expected output of `0`.

Remember that pointers hold an address and dereferencing a pointer attempts to read or store data at that address. This is what can lead you to access memory you should not be using. The following simple program is almost guaranteed to crash on any computer.

```
// this program is incorrect
int main()
{
  int *p;
  p = (int*) 8;
  *p = 1;
  return 0;
}
```

This program sets the pointer variable to the memory address `8`; we had to use a cast for the compiler to accept it since you should not set pointers directly to integer values (you should use the unary ampersand operator or `new` statement to set a pointer variable to a valid address). Executing the statement `p = (int*) 8` does not crash the program, but that is the incorrect line. The statement `*p = 1` attempts to store the value 1 at memory address 8 which is not part of the memory used for dynamic memory. The hardware detects this error and the program crashes. Again, you would not do this as part of a normal program, but this should show you that if you accidently set a pointer variable to an address that your program is no longer using and then later try to dereference that pointer, your program will crash. A more realistic example of the same problem is the following program.

```
// this program is incorrect
int main()
{
  int *x;
  *x = 5;
  return 0;
}
```

In this example, `x` is never initialized to hold a valid address so whatever address is already in the memory used for the variable `x` is the address in which the program will attempt to store the value `5`. If that address happens to be in one of the pages the operating system gave our program, the program will not crash. This is unlikely. It is more likely that the address stored in the variable for `x` is not a valid address so attempting to store a `5` there will cause the hardware to generate an exception and our program will crash. Here is another example that could cause the same problem.

```
int main() // this program is incorrect
{
  int *y = new int;
  delete y;
  *y = 3;
  return 0;
}
```

In this example, we attempted to dereference the pointer `y` and store the value `3` at that address after we have deallocated the memory location that `y` points to. This again could cause our program to crash or it could run to completion. You should be starting to see why these types of errors can be difficult to track down in larger programs.

We will examine one more program with errors in this chapter, but there are lots of different ways you can have these problems. This program has two errors.

```
// this program is incorrect
int main()
{
  int *x, *y;
  x = new int;
  y = new int;
  *x = 3;
  y = x;
  *y = 3;
  delete y;
  delete x;
```

The first problem is that this program has a memory leak. The memory for two integers is allocated, but then the statement `y = x` causes both pointers to refer to the same memory location. This results in the memory allocated by the statement `y = new int` being leaked since there is no way to access it and delete it. The `delete y` statement deletes the memory allocated by the `x = new int` statement. Since `x` also pointed to that memory location, the statement `delete x` attempts to deallocate the same block of memory a second time. This will likely corrupt the dynamic memory heap. This can also cause your program to crash immediately, or at a later time, or never.

10.5.3 Memory Error Summary

Some C++ run-time environments do not show you the exact line where your program crashed or a stack trace showing the function or method calls that resulted in the program crashing at that line. Most IDEs (integrated development environments) will show you the execution traceback similar to Python indicating at what line the program crashed and the functions or methods that were called to get to that point. This information is important for determining why your program crashed. Unfortunately, as we have discussed, the line your program crashed at is not necessarily the line that is incorrect. If the line it crashes at is dereferencing a pointer, the problem is that either you forgot to give that pointer a valid address or somehow it ended up pointing to memory that is no longer valid for your program to use (for example, you already called `delete` on that memory block or it got set to a value that does not correspond to a valid address). The traceback tells you the order the functions or methods were called to the point of the crashing. This helps you determine the code that caused the problem.

Also, as we mentioned earlier, sometimes you can corrupt the dynamic heap by accessing incorrect memory locations or calling `delete` twice for the same block of memory. This will typically not result in a crash until you try to allocate memory again. These types of errors can be extremely difficult and frustrating to track down. Fortunately, while you are developing your code, you can use an IDE that provides a debugger to help track down these errors. Debuggers provide a number of features to help you find errors in your programs. Most allow you to stop execution at specific source code lines within your program, examine the values of variables at that point, and execute one line or one function at a time while you watch the values of the variables. Debuggers typically provide additional capabilities beyond the ones we listed here.

When running your program within a debugger and your program crashes, the debugger will typically show you similar information to the Python traceback. It is fairly easy to develop Python code without a debugger, but when writing dynamic memory code in C++, a debugger and good IDE will help you track down memory errors more quickly and with less frustration. Sometimes proofreading the code around the crash (or for the entire class if the crash is in a method) is the most effective way to solve the problem.

It is always a good idea to find the smallest sample input that causes your program to crash or to work incorrectly. This is especially important when dealing with dynamic memory errors. If we determined that our `List` class did not work correctly and crashed in the `append` method, we should first check if it can happen when appending fewer items than cause the `resize` method to be called. If this is the case and we have only called the `append` method and the constructor, we know that the problem is with the constructor or `append` method. If it crashes in the `append` method only after the `resize` method has been called, then the problem could be in the constructor, `append`, or `resize`, but in this case we recommend checking the `resize` method first. Try to minimize the amount of code that is executed but still causes the problem. Limiting the amount of code you have to check will enable you to find the problem faster and with less frustration.