4

Arrays

Almost all computer programs deal with large amounts of data, and the amount is typically not known in advance. Sometimes this is hidden, sometimes more obvious. Consider a string of characters; what data is actually stored in a string object? Clearly, each character in the string must be stored, so there is a character number 1, character number 2, and so on. Based on our current knowledge, how would we represent this data? We could declare many character variables, `c1`, `c2`, and so on. This clearly has some major drawbacks: First, the length of any string we can handle will be limited by the number of variables; second, it will be very difficult to examine all of the characters. Suppose, for example, we have 100 character variables, and we want to know if any of them are the character `'a'`. We would like to use a loop to examine them, but we can't; there is no way to count through the variables. We'd like to do something like this:

```
//  This code does not work!
for (i=1; i<=100; i++) {
  if (ci == 'a') {
    cout << "Found the letter 'a'." << endl;
    break;
  }
}
```

but that will not work. C++ interprets `ci` as a separate variable, not as the variable formed by `c` followed by the value of integer `i`. In this situation we would have to type code to examine each of the 100 variables, one by one.

4.1 Using arrays

Most programming languages, including C++, provide a mechanism for manipulating large numbers of data items of the same type, the **array**. In fact, using arrays, we can write working code that is almost identical to the non-working code above. Here is how:

```
char c[100];
// Code here assigns characters to the array.
for (i=0; i<100; i++) {
  if (c[i] == 'a') {
    cout << "Found the letter 'a'." << endl;
    break;
  }
}
```

We start by declaring an array of 100 chars, called `c`. Then to refer to variable number `i` we use `c[i]`. The expression `c[i]` acts exactly like a variable of type char, but as desired we can specify which one we want by number instead of by a unique name. Note one other change: the values of `i` in this loop are 0, 1, 2,..., 99; every array in C++ is numbered starting at 0. The declaration `c[100]` gives us 100 variables of type char, numbered 0 through 99.

In practice, the "100" should not be typed explicitly—it is a magic number. The code should really look more like this:

```
#define MAX 100
char c[MAX];
// Code here assigns characters to the array.
for (i=0; i<MAX; i++) {
  if (c[i] == 'a') {
    cout << "Found the letter 'a'." << endl;
    break;
  }
}
```

If the array `c` contains a string of characters, it will typically not always be exactly 100 characters long; if we want to check the string for the presence of `'a'`, we want to examine only the characters that actually make up the string; here the variable `n` is assumed to be holding the length of the string; for example, if the string is 10 characters long, `n` is 10 and the characters are stored in `c[0]` through `c[9]`.

Note that this addresses only one of the two drawbacks mentioned above: it is now easy to examine the data with a loop, but we are still limited to strings of length 100. We have partially addressed this problem: it is easy to change the 100 to something else and recompile the program. What we would like is some way to change the array size as the program runs.

4.2 Arrays in classes

Arrays may be contained in a class, just as ordinary variables can. As an example, let us suppose that the string class did not already exist, and see how we might begin to make our own. Here is a first try:

```
#define MAX 100
class mystring {
private:
  char c[MAX];
  int lnth;
public:
  mystring();
  bool append(char x);
};
```

Here we have the array to hold characters and a variable `lnth` to indicate the number of characters, at most `MAX`, in the string. Initially the string should be empty; the array itself can't be "empty", but if the length is zero then the object represents the empty string. So the constructor should be:

```
mystring::mystring() {
  lnth = 0;
}
```

I have included one function, `append`; the intent is that this function will add one character to the end of the string. How will we do this? The length tells us where to find the first "unused" spot following the current characters in the string; we put the new character there and then add one to the length:

```
bool mystring::append(char x) {
  if (lnth < MAX) {
    c[lnth] = x;
    lnth++;
    return true;
  }
  return false;
}
```

Note that we check to make sure the array is not already full, then add the new character. If the array already contains MAX characters, we do not add the character and we return "false"; as long as the program checks this return value, it will know that something has gone wrong. This, of course, is where we would like to be able to increase the size of the array instead of simply failing to add the character.

4.3 Resizing arrays

We can't really resize arrays in C++, but we can do the next best thing: create a new array of a different length, then copy the data from the old array to the new one, and finally throw the old one away. To do this, we need to use a new C++ concept, that of a *pointer*. We start by rewriting the class definition:

```
class mystring {
private:
  char* c;
  int lnth, capacity;
public:
  mystring();
  ~mystring();
  bool append(char x);
};
```

We have now declared `c` to be a pointer to a char, and we have added a new variable, `capacity`, to keep track of the current length of the array. A pointer is like a street address; instead of holding a value directly, it gives the address in the computer's memory at which the value may be found. The value may be a lone value or the first value in a whole array, which is what we want to do here.

Merely declaring the pointer does not give it a reasonable value, nor does it reserve any space in memory to hold the array. For that we need to modify the constructor function:

```
mystring::mystring() {
  capacity = INITIAL_LENGTH;
  c = new char[capacity];
  lnth = 0;
}
```

After the constructor runs, `c` points to the beginning of an array of char values; the initial size, `INITIAL_LENGTH`, would be created using a `#define` line as usual. Now the basic operation of the append function is the same, but in case the array is full we want to create a new, longer array to replace the old one:

```
bool mystring::append(char x) {
  char* temp;
  if (lnth >= capacity) {
    temp = new char[2*capacity];
    for (int i=0; i<capacity; i++) {
      temp[i] = c[i];
    }
    capacity *= 2;
    delete [] c;
    c = temp;
  }
  c[lnth] = x;
  lnth++;
  return true;
}
```

There is quite a bit new going on here. First, we create a new, temporary array `temp`, using the same `new` operation as in the constructor function. Then we copy the existing values from `c` into `temp`. Next, we double the value of the variable `capacity` to reflect the new array length. The next line deletes the memory used by the old array `c`; "delete" is actually a misleading term, since the memory is still there, but it can now be reused for something else. The empty brackets `[]` tell C++ to delete an entire array of values, not just the first one. If you forget to delete memory when it is no longer in use, you have created a *memory leak*. In programs that run for a long time without being restarted, like a web server or even a web browser, memory leaks can cause the system to become sluggish or crash, as the computer runs out of memory.

We finish the if-statement by renaming the new array to `c`. This does *not* copy the actual characters in the array. Remember that a pointer is like an address; the statement `c=temp` is like a change of address. The old address in `c` is no longer valid, and the address of the new array is stored in `temp`. We copy the good address into `c` so that we can continue to use `c` as the correct address.

At this point, whether the code in the if-statement was executed or not, we know that there is room in the array for another character, and we add the new character as before.

Finally, there is another potential memory leak that we need to address. If a `mystring` object is declared in a function, then it is created when the function is called, and its memory is released to be reused when the function returns. If the object points to additional memory, as `c` does, C++ will not know to reclaim that memory as well, so we need to do it explicitly. Just as a constructor function runs when an object is created, a destructor function runs when the object is destroyed; this is the function `~mystring`. The destructor function must have the same name as the class with the tilde added at the beginning. Our destructor is very simple:

```
mystring::~mystring() {
  if (c) delete [] c;
}
```