Programming in C++

The original version of this text was written by David Guichard, guichard@whitman.edu.

This copy of the text was produced at 20:37 on 4/10/2014.

Contents

1

Basics

1.1 A first program

Every program we write will have these elements:

```
#include <iostream>
using namespace std;
int main () {
  cout << "Hello world!" << endl;
}
```

The first line is not literally required, but it will always be present in our programs; it tells C++ to include the capability to read input from the keyboard and write information to the screen. Likewise, the second line is not required, but should always be included. Many of the commands we will use to get the desired behavior from our programs are listed in a "namespace" called "`std`"; we must tell C++ that we plan to use commands from this namespace so that it can find them. (Think of a namespace as a kind of dictionary.)

Every C++ has a main function, which as shown consists of "`int main ()`" followed by matching braces "`{...}`"; between the braces are additional instructions or **statements**. In this simple example, the program is instructed to print "Hello world!" (without the quotes) on the screen; "`cout <<`" always means to print something following to the screen. (The command "`cout`" is in the standard namespace "`std`". If we had not told C++ we planned to use this namespace, we could not use `cout` in this way.) More than one piece of information can be sent to the screen with a single cout statement. In this example, following the message "Hello world!", an "end of line" command is sent to the screen. This causes the cursor on the screen to move to the beginning of the next line. Most statements in C++ are terminated by a semicolon.

How do we actually run or execute this program? C++ is a compiled language, meaning that before we can run the program we must translate it to "machine language," a form in which the computer can directly execute the program. Some standard conventions make this process quite simple. To identify the program above as a C++ program, we save it in a file with a ".cc" extension; suppose we have stored it in "`hello.cc`". Then in a shell window we need only type "`make hello`". The make program will look for a file named "`hello.something`", and if it finds one, will check to see if it knows how to make `hello` from `hello.something`. If it does, it proceeds to do so. The program that makes "`hello`" from "`hello.cc`" is the C++ compiler, "`g++`", so make will automatically invoke the command "`g++ hello.cc -o hello`". You could type this command yourself instead of the simpler "`make hello`". The "`-o hello`" tells the `g++` command to put the output, which is the compiled program, into a file called "`hello`"; this is the program that we can actually run. We can illustrate the compile process like this:

$$\boxed{\texttt{hello.cc}} \xrightarrow{\texttt{make hello}} \boxed{\texttt{hello}}$$

Note well that every time you make a change to a C++ program file, like `hello.cc`, you must save the changes and then run the `make` command again to create a new executable program.

Now in a shell window, we compile and run the program like this:

```
> make hello
g++     hello.cc   -o hello
> ./hello
Hello world!
>
```

The shell knows to look in a few standard places for commands, but it normally does not look in the current directory; by typing "`./hello`" we tell the shell that it is in the current directory; "`./`" is a shorthand expression that always means the current directory.

1.2 Variables

A program that does exactly the same thing every time it is run is not very useful. Variables in a program can assume different values at different times, and the program can then produce different results, depending on circumstances. It is not enough to have variables, however, we also need a way for the program to accept input when it runs, so the values of variables are not fixed in advance. While programs can receive input in a variety of ways, our simple programs will allow the human user to provide input from the keyboard.

As an example, let's look at a program to perform a simple calculation, converting feet to meters. Here's a start:

```
// Convert feet to meters
#include <iostream>
using namespace std;

int main () {
  double distance;

  cout << "Enter distance in feet: ";
  cin >> distance;
  cout << distance * 0.3048 << endl;
}
```

In C++, every variable has a **type** that describes what sort of value it can contain. In this program we have one variable, `distance`, of type `double`. A "double" value is a number that may have both a whole number part and a fractional part, like 13.7 or 17.0. The first cout statement is familiar. The statement "`cin >> distance`" is an **input** statement; when the program reaches this point, it will stop and wait for something to be typed at the keyboard; when the enter or return key is pressed, the typed information will be given to the program, and the value will be stored in the variable `distance`. Finally, on the last line, we use another cout statement to print the value of `distance` multiplied by 0.3048, which is the number of meters in one foot. This shows that we can use cout to print numerical values, and can compute the value as part of the cout statement.

This program also includes a **comment**, that is, some text that is not part of the program but is included to provide information to human readers. Any text following "`//`", to the end of the line, is ignored by the compiler. Comments can be used to include identifying information, like the name of the programmers who worked on the code, and explanatory information like what a section of code does and how. Ideally the "how" is adequately explained by the code itself, but some code is inherently hard to understand, and then explanatory comments are a good idea.

Variables can hold many types of data, from simple numerical values to complicated sets of data. Initially we will need only simple data types, including:

- `double` A numerical value, as we have seen.
- `int` A numerical value that is always an integer, that is, a positive or negative whole number, or zero. While it is often possible to use a double variable even when we know the value will always be an integer, there are advantages to using variables that can only be integers, as we will see.
- `char` A char variable holds a single character, like 'a' or '/' or '.'. Explicit char values are typed between single quotes.
- `bool` A bool variable may have one of two values, "`true`" or "`false`". True and false values are enormously useful for writing programs that change their behavior depending on circumstances, that is, depending on whether certain statements about values are true or false. "bool" is short for "boolean", the standard term for true and false in computer science and mathematics.

Variables by themselves are not particularly useful; we need to be able to combine and alter values to accomplish a task. In the last program we already did this when we multiplied two quantities together. C++ includes many operations on variables; here are a few of the simplest and most useful.

First, the assignment operator "`=`" works on variables of any type. If `d` is a double variable, then "`d = distance * 0.3048`" assigns to `d` the value of the expression to the right of the operator. The principal duty of the assignment operator is to assign a value, but it is also an operator in the more familiar sense: it produces a value. The value of `a=5` is 5; the statement "`b = 3 * (a=5)`" assigns `a` the value 5 and `b` the value 15. A more common use is "`a = b = 1`", which assigns `b` the value 1, then assigns that value (still 1) to `a`.

Next we have the familiar arithmetic operators, and some unfamiliar ones that are particularly useful:

`+` Addition of two numerical quantities, e.g., `a+b`.
`-` Subtraction, e.g., `a-b`.
`*` Multiplication, e.g., `a*b`.
`/` Division, e.g., `a/b`.
`++` Increment, e.g., `n++` adds 1 to `n` and saves the new value in `n`.
`--` Decrement, e.g., `n--` subtracts 1 from `n` and saves the new value in `n`.
`+=` Add to, e.g., `n += 5` adds 5 to `n` and saves the new value in `n`.
`-=` Subtract from, e.g., `n -= 5` subtracts 5 from `n` and saves the new value in `n`.
`*=` Multiply by, e.g., `n *= 5` multiplies `n` by 5 and saves the new value in `n`.
`/=` Divide by, e.g., `n /= 5` divides `n` by 5 and saves the new value in `n`.

These operations can be used on variables and on explicit values, and may include variables of both `int` and `double` type. For example, here is a bit of legal code:

```
double a;
int b;
cin >> a >> b;
cout << a*2 + b - 3.2*a*b << endl;
```

The operations of C++ have a precedence ranking, meaning the order in which they are done depends both on the order in the expression and the relative precedence. For the simple arithmetic operators, the rules are already familiar to you: multiplication and division are done first, addition and subtraction after that; within a precedence level, operators are applied left to right. As you would expect, this order can be altered by using parentheses in the usual way.

When values of integer and double type are mixed, the results can be a bit surprising until you know the rules. Since every integer value is also a double value, there is generally no question about the value, but the type of the value is also important. As C++ computes the value of a complicated expression, any portion of the expression that consists solely of integer values is also an integer value; as soon as a double enters the calculation, all values are converted to double. Consider, for example, "`a*2+b*b`" in the previous context. C++ performs the multiplications first: "`a*2`" is a double times an int, so it is a double value; "`b*b`" is an int. When the double is added to the int, the result becomes a double.

While the actual value is usually not in question, there is an exception: consider "`b/10`", where again `b` is an int. Since both `b` and 10 are ints, the result is an int. If, for example, `b` has value 23, then `b/10` has value 2, the ordinary whole number quotient. The fractional part, if any, is discarded. This can be quite useful, but it can also lead to incorrect results if you forget the rule. It also means that order matters in a way you are not familiar with. For example, "`b/10*a`" and "`a*b/10`" will in general have different values, because in the first, `b/10` will be an int, discarding the fractional part. You can force integer values to be interpreted as doubles in two ways. For explicit values, like 10, put in the decimal point: 10.0. (You can also write just "10.", but that is a bit harder to read and understand.) For both explicit values and variables, you can **cast** an int to a double, like this: `(double)b` or `(double)10`. Thus, all of these have the same value:

```
a*b/10
b/(double)10*a
(double)b/10*a
b/10.0*a
```

When doing "integer division", in which 23/10 is 2, there is of course the accompanying notion of the remainder. This is often quite useful, and C++ contains an operator to compute the remainder directly: "`%`". For example, `23%10` is 3; this is usually read "23 mod 10".

There are three operators on boolean values that are frequently used. Remember that the only boolean values are true and false; the boolean operators combine these to produce true or false.

`&&`	The "and" operator. "`true && true`" is `true`, all other combinations are `false`.
`||`	The "or" operator. "`false || false`" is `false`, all other combinations are `true`.
`!`	The "not" operator. "`!true`" is `false`, and "`!false`" is `true`.

Boolean values are most often used in the context of operators that produce boolean values but operate on other values. The most common are:

`<`	Less than
`<=`	Less than or equal
`>`	Greater than
`>=`	Greater than or equal
`==`	Equal
`!=`	Not equal

Since these operators produce boolean values, they can be combined with boolean operators, for example: "`a!=2 && ( a<-1 || a>1 )`".

Let's return to our last program and modify it a bit.

```
#define METERS_PER_FOOT 0.3048
#include <iostream>
using namespace std;
int main () {
  double distance_feet, distance_meters;
  cout << "Enter distance in feet: ";
  cin >> distance_feet;
  distance_meters = distance_feet * METERS_PER_FOOT;
  cout << distance_meters << endl;
}
```

This does the same thing as before, but the changes illustrate some new points. First, instead of using 0.3048 without explanation, we have given it a self-explanatory name, and then used the name in the calculation. The value 0.3048 is a "magic number"—a special value whose significance is not obvious. By giving it a name, we explain the value, we make the code that uses it easier to understand, we make it less likely that we will mistype 0.3048 in some instances (if we were to use it repeatedly in a long program), and we make it easy to change the value everywhere by changing it just once. (For example, we might later decide that we need 6 decimal places of accuracy instead of 4.) All special numbers should be treated this way.

It is important to understand that in this context, METERS_PER_FOOT is not a variable, so it would not make sense to use METERS_PER_FOOT = 5 in the program. METERS_PER_FOOT is a **preprocessor symbol**. Before the program is compiled, a preprocessor runs first. Every instance of METERS_PER_FOOT in the program is replaced by 0.3048, just as if you had typed it. The resulting program is then compiled. Thus, by the time the compiler runs, METERS_PER_FOOT = 5 has become 0.3048 = 5, which doesn't make sense. Any line that starts with “#” is a preprocessor directive. When we use “#include <iostream>”, the preprocessor finds a file called iostream and includes it in the program, again just as if we had typed it. In short, compiling is more complicated than indicated on page 6; the picture is actually:

$$\boxed{\texttt{hello.cc}} \xrightarrow{\texttt{preprocess}} \boxed{\texttt{....}} \xrightarrow{\texttt{compile}} \boxed{\texttt{hello}}$$

The preprocess stage does not produce an actual file; the output goes straight to the compiler.

Next, by introducing two variables for the two distance values, we clarify the code and give ourselves more specific variables to work with if we expand the program to do something more complicated. By first assigning the new value to a variable, distance_meters, we are able to print the value as before, but we retain the value for further use if required.

1.3 Functions

We now rewrite the previous program in yet another way:

```
#define METERS_PER_FOOT 0.3048
#include <iostream>
using namespace std;

double feet_to_meters (double x) {
  return x*METERS_PER_FOOT;
}

int main () {
  double distance_feet, distance_meters;

  cout << "Enter distance in feet: ";
  cin >> distance_feet;
  distance_meters = feet_to_meters(distance_feet);
  cout << distance_meters << endl;
}
```

Here we have introduced a function, called feet_to_meters, which is essentially a subprogram. As programs get longer, the proper use of functions can help in two principal ways. First, it can help to organize the code into logical "modules", which makes it easier to keep track of what you are doing. Imagine that the conversion of feet to meters involved many complicated calculations. The line "`distance_meters = feet_to_meters(distance_feet);` would not change, and would still be easy to understand: "Here we go away, do a particular conversion, then come back." While the function may be more or less complicated, the main program (which is itself a function—the main function) is the same. Second, in a long program, a particular task may need to be done repeatedly. By putting the code to accomplish this task in a function, we escape typing out all the code every time we need to accomplish the task. This makes the program shorter, easier to read, and less prone to error.

While functions do "the same task" when **called** (that is, invoked), they can be provided with data via **parameters**, so that they perform a similar but not identical computation. In our example, we want the function to convert feet to meters, but we want to be able to specify the number of feet when the program runs. When the function is **declared** as `double feet_to_meters (double x)`, we are telling C++ that when the function is called we will provide a double value. This double value should be assigned within the function to the variable `x`, and then `x` will be used in the calculation. So when we call the function as `feet_to_meters(distance_feet)`, first `distance_feet` is copied into `x`, then the function runs to compute `x*METERS_PER_FOOT`. Finally, this value will not be useful in the main function unless the computed value is returned to the point at which we need to use it—in this case, we want to copy the computed value into `distance_meters`. The "`double`" before the function name in the declaration `double feet_to_meters (double x)` indicates that this function will return a double value. Inside the function, we specify what value to return by using the `return` statement. When we then call the function in context, like `distance_meters = feet_to_meters(distance_feet)`, the effect is the same as if we had typed `distance_meters = distance_feet * METERS_PER_FOOT`. Sometimes you will not want to return a value from a function; in this case the return type should be declared as `void`.

One point bears emphasizing: when we call `feet_to_meters(distance_feet)`, the value of `distance_feet` is copied into `x` and the calculation inside the function is done with `x`. If during the course of the calculation the value of `x` changes, this does not affect the value of `distance_feet`. Consider this function, for example:

```
void increment (int n) {
  n = n + 1;
}
```

If we call this function as `increment(x)`, the value of `x` does not change. Sometimes we would like it to change; in this case, the whole point of `increment(x)` is presumably to add one to `x`. We can write a function that will accomplish this:

```
void increment (int& n) {
  n = n + 1;
}
```

The simple addition of the "`&`" does the trick. Now when we call `increment(x)`, `x` itself is used inside the function in place of `n`.

While of course functions should do something useful, they should not do too much. In particular, functions are most useful when they do a single task. In our simple example, we could have had the function both compute and print the distance in meters, but that would limit the function's usefulness. It is not hard to imagine that you might want to convert a distance from feet to meters without displaying it on the screen. Write functions that do a single task; in particular, a single function should not do both computation and output (or input).

2

Making Choices

2.1 If statements

The if-else statement is the simplest but most general mechanism for making a choice of statements to execute. In outline it looks like this:

```
if ( <boolean value> ) {
  statements
} else {
  statements
}
```

Here `<boolean value>` is exactly that: any expression that produces a true or false value. The "statements" are ordinary C++ statements. Here is a simple example:

```
if ( n<0 ) {
  cout << "negative";
} else {
  cout << "positive or zero";
}
```

If the boolean expression is true, the statements in the "if-part" are executed and those in the "else-part" are not; if it is false, the if-part is skipped and the else-part is executed. The else part is optional:

```
cout << "You have " << n << " day";
if ( n>1 ) {
  cout << "s";
}
cout << " left until your card expires." << endl;
```

If `n` is bigger than 1 the if statement will add an "s" to "day", otherwise it will do nothing.

The statements inside the if statement can be any legal C++ statements, including if statements:

```
if ( n<0 ) {
  cout << "negative";
} else {
  if ( n>0 ) {
    cout << "positive";
  } else {
    cout << "zero";
  }
}
```

The braces surrounding the if-part and the else part are optional if there is a single statement in the part, so this is legal:

```
if ( n<0 )
  cout << "negative";
else
  if ( n>0 )
    cout << "positive";
  else
    cout << "zero";
```

Note that the "inner" if-else-statement counts as just a single statement. It is generally a good idea to put in the braces even if they are not needed. Then if you modify an if-statement by adding one or more statements to the if-part or else-part, everything will be fine. If you leave out the braces at first, and you forget to add them when you add new statements, the code will not work as intended. The braces often make the code easier to understand as well.

2.2 Repetition

C++ provides three **loop** statements that perform a collection of statements repeatedly. The loops are to some extent interchangeable, but each is more appropriate in some circumstances. If you were to pick just one to use always, it would probably be the for-loop:

```
for ( <initialize> ; <continue condition> ; <update> ) {
  statements
}
```

The for loop will do the "statements" repeatedly; the three fields in parentheses control the number of repetitions. When the program starts to execute the for-loop, it first executes the "initialize" statements listed before the first semicolon; these are done only once. Most often there will be a single statement here; if more than one, they are separated by commas. Next, the "continue condition" is checked. This is a boolean value; if it is true, the statements in the body of the loop will be executed, otherwise the for-loop terminates. After the statements in the body have been executed, the "update" statements are executed; again, most often there will be only one, and multiple statements are separated by commas. Now the process repeats: if the continue condition is true, the statements in the body are executed and the update statements are executed. Here is a simple example:

```
for ( i=1,j=n ; i<=n ; i++,j-- ) {
  cout << i*j << endl;
}
```

First, `i` is set equal to 1 and `j` to `n`. Then the code repeatedly computes and prints `i*j`, each time increasing `i` by 1 and decreasing `j` by 1. This continues until `i` is larger than `n`. The numbers printed will thus be $1 \cdot n$, $2(n-1)$, $\ldots$, $(n-1)(2)$, $n \cdot 1$.

Any of the three fields in parentheses may be left blank, but there must be exactly two semicolons. If the first field is blank, there simply are no initial statements, and likewise if the third is blank, no statements are executed other than the statements listed in the body. An empty second field is interpreted as true, so the loop will run forever unless a statement inside the for-loop causes it to stop. This code does the same thing as the previous example:

```
i=1; j=n;
for ( ; ; ) {
  if ( i>n ) { break; }
  cout << i*j << endl;
  i++; j--;
}
```

The `break` statement immediately terminates the loop. While this code accomplishes the same task, it is not as easy to understand (and would be even harder if it were longer and more complicated). By placing the initialize statements, the continue condition, and the update statements in the for-loop's first line, we make clear what sort of repetition the loop will produce.

Note that it is possible that the statements in the for-loop will not be executed at all, if the continue condition is false at the beginning. In the first example, if `n` is zero or negative, the condition `i<=n` will be false after initializing `i=1`, and so the for-loop will terminate immediately.

The while-loop is essentially a for-loop in which the first and third fields are blank. This while loop does the same thing as the previous for-loops:

```
i=1; j=n;
while ( i<=n ) {
  cout << i*j << endl;
  i++; j--;
}
```

Because the inialize and update statements are still here, this is still not as readable as the original for-loop. In some situations, there really are no initalization statements, and the update statements may be different depending on the circumstances each time the body of the loop is executed. Ultimately, whether to use a for-loop or a while-loop is a matter of preference.

The for-loop and the while-loop both check a boolean condition before executing, so that they can execute zero times. Occasionally you know that you want to do the loop body once in all cases, after which you want to continue with normal loop behavior. Doing this with a for-loop or while-loop typically means that you type the body of the loop before the loop, to do it the first time, and then repeat it in the body of the loop. An alternative is the do-while loop:

```
i=1; j=n;
do {
  cout << i*j << endl;
  i++; j--;
} while ( i<=n );
```

This behaves just like the previous loops except when `n` is zero or less, in which case it computes and prints $1 \cdot n$ and then quits, while the previous examples do nothing. Here is a more interesting example:

```
do {
  cout << "Enter distance in feet: ";
  cin >> distance;
  cout << "That is  " << distance * M_PER_F << " meters." << endl;
  cout << "Do another? ";
  cin >> answer;
} while (answer != 'n');
```

Here `answer` is a char variable; typing "y" (or in fact anything other than "n") in answer to "Do another?" will keep the loop going.

2.3 Switch statements

Sometimes we will need to choose an action based on the value of a variable, and we know in advance what the possible values are, and that there are a small number of them. For example, we may present a menu of choices to the user, and we need to take action based on which option the user chooses. We can, of course, do this with if-statements, something like this:

```
if (answer == 'a' || answer == 'A') {option A statements}
else if (answer == 'm' || answer == 'M') {option M statements}
else if (answer == 'q' || answer == 'Q') {option Q statements}
else cout << "Error" << endl;
```

While there is nothing wrong with this, the **switch** statement has some advantages. Here is a switch that does the same thing as the code with if-statements:

```
switch (answer) {
  case 'a':
  case 'A': option A statements; break;
  case 'm':
  case 'M': option M statements; break;
  case 'q':
  case 'Q': option Q statements; break;
  default:  cout << "Error" << endl;
}
```

Based on the value of `answer`, the switch will jump directly to the matching case and perform the statements following the colon. This is more efficient than the if-statements, because of the direct jump—the switch doesn't repeatedly check every possible value. The switch statement also tends to be easier to read and understand, especially if the code in each case is short (and it can always be made short, by calling a function). After the switch jumps to the correct starting point, all the statements after that point will be executed, even the code for other cases, unless a `break` is encountered. In this case, a break stops the switch statement. The example code uses this feature: if `answer` is `'a'`, the switch jumps to the `'a'` case. Since there is no `break` in that case, execution continues to the `'A'` case, so the statements there are executed, and then the break causes the switch to end. If `answer` matches none of the cases, the default case is executed if present—the default case is optional. If `answer` matches no case and the default is missing, then the switch statement does nothing.

The switch statement does have some limitations. The case values may only be integers or characters, and there is no way to abbreviate a range of values—you can't type something like `1..10`, for example; you must type out all of the values from 1 through 10. Sometimes you can combine an if-statement with a switch statement for the most efficient and readable solution, perhaps something like this:

```
if ( ( 'a' <= answer && answer <= 'z') ||
     ( 'A' <= answer && answer <= 'Z') ) {
  // Do something with letters.
} else if ( '0' <= answer && answer <= '9' ) {
  // Do something with numbers.
} else {
  switch (answer) {
    case '+': plus statements; break;
    case '-': minus statements; break;
    case '*': times statements; break;
    case '/': divide statements; break;
    default:  cout << "Error" << endl;
  }
}
```

3

Classes and objects

Most of what we have seen so far is really the programming language C. C++ is a language that extends C in one important way: it adds classes and objects; C++ is an "object-oriented" language.

A class is a type of data, like int and char and bool, but typically more complicated, together with some operations on the data. An object is a particular instance of a class, usually created by declaring a variable whose type is the class. Think of a class as a way to "package" some data and some functions for manipulating the data into a single object. This encourages the sort of modularization that is good programming practice in general, and it makes it easier to reuse code in new programs.

We have already used two class objects, `cout` and `cin`; `cout` is an object of type `ostream` and `cin` is an object of type `istream`. Because our use so far has been quite simple, the properties of these objects have not been fully apparent.

3.1 Strings

Let's explore the idea by looking at a particular class, the string class. A string is a sequence of characters, like a word, or a sentence, but in general any sequence of values of type char. The data in a string object is this sequence of characters, plus some other auxiliary data, like the length of the string. The string class also includes many functions for manipulating the string or accessing properties of the string. You can get information about all of the functions at

`http://en.cppreference.com/w/cpp/string/basic_string`.

Let's look at a simple program to see how to use a class object.

```
// Convert phone number xxx-xxx-xxxx to (xxx) xxx.xxxx
#define LOCAL_AREA_CODE "509"
#include <iostream>
#include <string>
using namespace std;

int main () {
  string phone,area_code,prefix,local;
  cout << "Enter a phone number: ";
  cin >> phone;
  string::size_type hyphen1 = phone.find('-');
  string::size_type hyphen2 = phone.find('-',hyphen1+1);
  if ( hyphen2 != string::npos ) {
    area_code = phone.substr(0,3);
    prefix = phone.substr(hyphen1+1,3);
  } else {
    area_code = LOCAL_AREA_CODE;
    prefix = phone.substr(0,3);
  }
  local = phone.substr ( phone.find_last_of('-')+1);
  cout << "(" << area_code << ") " << prefix << "." << local << endl;
}
```

Here's what's going on:

1. We use `#include <string>` to load the information about the string class that C++ will need.
2. We declare four string variables, one for the entire phone number that we will read, and three for the three different parts of the phone number.
3. We can read strings directly from cin. When we do, C++ first skips over any blank space, then reads characters until it finds more blank space. Even though spaces are characters, we cannot read them in this way.
4. Some variables and types used by the string class are not in the standard namespace; we access these by prefacing the name with "`string::`"; `size_type` is essentially an int, used for various purposes by the string class. We set `hyphen1` to the position in the string at which the first hyphen occurs, using the find function. The syntax `phone.find` means to use the find function inside the string `phone`. We next find the second hyphen; this time there is a second parameter, `hyphen1+1`, meaning we want to start looking for the second hyphen at the character following the first hyphen.

5. We are going to allow the input number to include an area code or not. If it does not, there is only one hyphen in the number; in this case `hyphen2` will be the special value `string::npos`, "not a position", meaning the desired character was not found. We can test for this to act appropriately in the two cases.
6. The function `substr(m,n)` extracts a substring from a string, starting at position m and consisting of n characters. The position numbering starts at 0, so `area_code = phone.substr(0,3)` extracts the first three digits, either the area code or the exchange prefix.
7. We already know where the hyphens are, but as a further example, we use `phone.find_last_of('-')` to find the last hyphen, which could be the second or the only hyphen. Then we extract the substring starting one character after this hyphen; since we do not specify the length, the `substr` function will extract all the way to the end of the string. This means that if we enter "509-527-5000x234", the output will be "(509) 527.5000x234".

There are other approaches to this problem. Using other string functions, we could have altered the string `phone` itself, inserting a "(" at the beginning, replacing the first hyphen with ") ", and replacing the final hyphen with a period. For example, to replace the last hyphen with a period:

```
phone.replace(phone.find_last_of('-'),1,".");
```

This replaces 1 character, starting at `phone.find_last_of('-')`, with the string in double quotes. In general, the replacement string could be either longer or shorter than the characters being replaced. To replace the first hyphen with two characters, a parenthesis and a space:

```
phone.replace(phone.find('-'),1,") ");
```

Here is a sample of the program in action:

```
> ./test
Enter a phone number: 656-123-3445
(656) 123.3445
> ./test
Enter a phone number: 123-3445
(509) 123.3445
```

3.2 New Classes

Many classes have been written to store and manipulate many types of data. Some, like the string and iostream classes, are available in most versions of C++. Others may be found free on the web or purchased from software companies. But sooner or later, if you program in C++ very much, you will need to write a new class. Writing new classes will occupy most of the rest of our time.

There are two sorts of entities in a typical class, data and functions. In the string class, the data consists of the actual characters of the string; it may be that the string class has other data as well, such as the current length of the string. Note that we are able to successfully use the string class without having a detailed knowledge of the data. This is true of many classes and is generally an advantage. As long as we can manipulate the data in a useful way, we need not be concerned with the details of how the data is stored. When we write a new class, of course, we do need to think about these details. Fortunately, the data in a class is created in much the same way that we have created data already, typically by declaring a variable with some data type like int. Likewise, creating a function "inside" a class is much like declaring a function in a program.

Let's begin by looking at a simple example. Suppose we wish to write a program that stores and manipulates lengths. We could simply use a variable of type double for each length we need, but by creating a length class, we can provide a more flexible and useful version. In particular, we can easily imagine that we would want to be able to manipulate lengths in different units. While the same length is represented by different numbers in different units, there really is just one real length. So we will create a class that holds just one value, the length, but that allows us to manipulate it in different units. How might we want to use such a quantity?

We presumably want to be able to create a variable to hold a length, something like:

```
length x,y,z;
```

Then we might want to perform a computation. Suppose that x, y, and z are the length, width, and height of a box. How might we compute the volume of the box in, say, cubic centimeters? It would be convenient to be able to do something like this:

```
cout << "Volume: " << x.cm() * y.cm() * z.cm()
     << " cubic centimeters" << endl;
```

This is not perhaps the first thing you would think of, but it matches what we already know about classes. To perform a function on the data in class object `x` we use the period notation followed by a function name. In this case, we want to have a function that will convert the length to centimeters. We can create a simple length class like this:

```
class length {
  private:
    double d;
  public:
    double cm();
    void cm(double x);
    double inches();
    void inches(double x);
};
```

The idea is that `d` will contain the length, and the functions `cm()` and `inches()` will report the length in different units. The other functions are used to set the value of `d`. If we want the length stored in an object to be 5 centimeters, for example, we will use something like `x.cm(5)`. Note that we have used the same names, `cm` and `inches`, for two different functions. Because one has no parameters and the other has a double as parameter, C++ can tell them apart. In general, you may have many functions with the same name as long as C++ can distinguish them by the number and types of the parameters.

What is the actual value of `d`? It holds the "true" length, but any length must be expressed in some units. We need to pick the units for `d`, then write the functions to correctly convert as needed. Let's say we record `d` in meters. Now we can write the functions. This is just like writing a function in a program, except that we somehow need to tell C++ that the function belongs in a class. Here's how:

```
double length::cm() {
  return d*100;
}
```

To report the length in centimeters, we multiply the length in meters by 100 and return that. Notice that the variable `d` is not declared in the function; C++ will find it in the length class because we have prefaced the function name with "`length::`". This tells C++ that this function is a part of the length class. The other centimeter function sets the value of `d`.

```
void length::cm(double x) {
  d = x/100;
}
```

The two functions for inches are nearly identical, but we need to replace 100 by the correct conversion factor. If we are actually going to write this class, it probably makes sense to include more units in the list, perhaps meters, feet, yards, and millimeters, at least.

This class will be useful in any program that manipulates lengths. In particular, we can easily use it to write a program to convert between lengths. We tell C++ that we plan to use strings by including the string class. Since we are writing our own class, we need to tell C++ about it directly. The simplest way to do this is to put the class definition and functions right in the program file.

```
// Convert between inches and cm
#include <iostream>
using namespace std;
#define INCHES_PER_METER 39.3700787
class length {
  private:
    double d;
  public:
    double cm();
    void cm(double x);
    double inches();
    void inches(double x);
};

double length::cm() { return d*100; }
void length::cm(double x) { d = x/100; }
double length::inches() { return d*INCHES_PER_METER; }
void length::inches(double x) { d = x/INCHES_PER_METER; }

int main () {
  length D;
  double x;
  char choice;
  cout << "Choose units to enter: (c) or (i): ";
  cin >> choice;
  cout << "Enter value in ";
  switch(choice) {
    case 'c': cout << "centimeters: "; break;
    case 'i': cout << "inches: "; break;
  }
  cin >> x;
  switch(choice) {
    case 'c': D.cm(x); break;
    case 'i': D.inches(x); break;
  }
  switch(choice) {
    case 'c': cout << x << " centimeters is "
                   << D.inches() << " inches." << endl;
              break;
    case 'i': cout << x << " inches is "
                   << D.cm() << " centimeters." << endl;
              break;
  }
}
```

To use the length class in another program, note that it is easy to identify which code you need to copy: the class definition, all the functions that start with “`length::`, and the conversion factors if, as in our code, they are given names. While this is not terribly difficult for small numbers of classes in a few programs, we can make it even easier. We already know that to use the string class we need only put "`#include <string>`" in a program. We can do something similar to this with our own classes.

The simplest way to do this is to put all of the class code in a separate file, perhaps called "`length.cc`". Then we replace the code in the main program file with "`#include "length.cc"`. When the program is compiled, the file `length.cc` will be inserted just as if it had been typed, as in the original version.

For small programs, this is a satisfactory solution. For larger programs, there is a better way, and it is easy enough that it is best to use it in all cases, for consistency. It turns out that to compile the main program, C++ really only needs the definition of the class, not the actual code for the functions of the class. Of course, at some point the functions must be compiled, but there is no reason to compile them every time the main program is compiled. In large programs there may be dozens or even hundreds of classes in use and many thousands of lines of code. Compiling them all each time the main program is compiled will slow the process down. When we use the string class, we do not compile all the string functions each time our program is compiled.

It turns out that what we have referred to up to now as *compiling* is really two separate steps: *compiling* and *linking*. In the compiling stage, our C++ code is turned into machine code; in the linking stage, this compiled code is combined with previously compiled code for all the classes, functions, and operations we have used. Many of the standard features of C++ are linked in automatically, and so we have not had to know this before.

To set our new length class up for this separate compilation, we need to create two files for the class information. In one, `length.h`, we put the "header" information:

```
// File length.h
#define INCHES_PER_METER 39.3700787
class length {
  private:
    double d;
  public:
    double cm();
    void cm(double x);
    double inches();
    void inches(double x);
};
```

In the other, `length.cc`, we put the actual functions:

```
// File length.cc
#include "length.h"
double length::cm() { return d*100; }
void length::cm(double x) { d = x/100; }
double length::inches() { return d*INCHES_PER_METER; }
void length::inches(double x) { d = x/INCHES_PER_METER; }
```

Note that we include the header file here. In the main program file, we include only the header file:

```
// File conversion.cc
// Convert between inches and cm
#include <iostream>
using namespace std;
#include "length.h"

int main () {
  length D;
  double x;
  char choice;
  cout << "Choose units to enter: (c) or (i): ";
  cin >> choice;
  cout << "Enter value in ";
  switch(choice) {
    case 'c': cout << "centimeters: "; break;
    case 'i': cout << "inches: "; break;
  }
  cin >> x;
  switch(choice) {
    case 'c': D.cm(x); break;
    case 'i': D.inches(x); break;
  }
  switch(choice) {
    case 'c': cout << x << " centimeters is "
                   << D.inches() << " inches." << endl;
              break;
    case 'i': cout << x << " inches is "
                   << D.cm() << " centimeters." << endl;
              break;
  }
}
```

Now we need to separately compile the two code files and link them together to form the executable program:

```
> g++ -c conversion.cc
> g++ -c length.cc
> g++ -o conversion conversion.o length.o
```

This is a bit more complicated than "`make conversion`"; here's what is going on: The first two lines compile the code from `conversion.cc` and `length.cc` into files `conversion.o` and `length.o`. The "`-c`" tells the compiler not to try to link anything yet, but to compile only. The final command links the two files together and puts the result in a file called `conversion` which we can run as usual. If the code in `length.cc` is already known to be working, and we are working on the code in `conversion.cc`, we do not need to compile `length.cc` more than once; all we need to repeat are the first and third commands as we make changes to `conversion.cc`.

This is all a bit tedious, especially if there are more classes involved, each class in its own file. In fact, with just a bit more work we can arrange to go back to typing "`make conversion`" but keep the advantages of separate files. We create one more file:

```
# File Makefile
conversion: conversion.o length.o
	g++ -o conversion conversion.o length.o

conversion.o: conversion.cc length.h
	g++ -c conversion.cc

length.o: length.cc length.h
	g++ -c length.cc
```

One feature of this file is not obvious: the indentation before each of the g++ lines *must* be done with a tab, not with spaces. Once this file has been created, we can type simply "`make conversion`" or even just "`make`" to compile and link the program. The make command is smart enough, given this Makefile, to compile only the code that needs to be compiled. If, for example, we have made a change to `conversion.cc` but not to `length.cc`, then only `conversion.cc` will be recompiled. Because we have listed `length.h` on the "dependency" lines for both `conversion.o` and `length.o`, if we make a change to `length.h` then both cc files will be recompiled. The reason we need only type "`make`" is because the first entry in this file is for the final program `conversion`; the make command alone will make the first item.

You will have noticed the labels "private" and "public" in the class definition. Any items in the private section, data or functions, are not available for use in a main program. If the variable `d` were in the public section, it would be possible in the main program to do this, for example: `x.d = 50`. In general, *all* data items should be in the private section, so that access to the data can be controlled. In the length class, for example, we ensure via the functions that the value `d` always represents meters, but the code in the main program does not need to "know" this, and indeed certain sorts of errors become difficult or impossible to make because the main program cannot directly access `d`. Moreover, if for some reason we wanted to change the units for `d`, say to feet, we could do this by rewriting the functions inside the class, and the code in the main program would not need to change at all.

In the length class, all functions are public. In some classes it is appropriate to put some functions in the private section, "auxiliary" functions that are useful within the class but that should not be used by the main program. The functions in the length class are all of two special types, called *reader* and *writer* functions. The two functions without parameters are reader functions, as they allow the main program to read the data stored in a class object. The other two are writer functions, which allow the main program to write values into the object.

3.3 Constructors

Our last program began like this:

```
int main () {
  length D;
  double x;
  char choice;
```

What are the values of `D`, `x`, and `choice` at this point in the program? We have not given them any values yet, but they must have values, since each variable is associated with some part of the computer's memory, and each memory location has some value. These variables are *uninitialized*, which means their values might be anything. Some compilers initialize some variables automatically; there is a good chance that `x` has the value zero, and that `choice` does as well, meaning it stands for the character with code zero. `D` contains one piece of data, a double, and it might have value zero (on my computer it does not). In any case, no compiler is required to initialize variables.

Does it matter? After all, in this program, we never attempted to use a variable before we had explicitly assigned a value to it. It is easy to become lazy, however, when you know that variables tend to be initialized in some suitable way. If you take your code to a different computer with a different compiler that does not initialize variables automatically, then your previously working program may mysteriously break. As a general rule of thumb, all variables should be explicitly initialized. With simple types, like double and char, this is easy to do, like this for example:

```
int main () {
  length D;
  double x = 0;
  char choice = 'x';
```

How do we initialize `D`? We can't use `D=0` because `D` and `0` are completely different types of value. As our classes become more complicated, and contain more data items, it becomes even clearer that something like `D=0` can't possibly work. Classes in C++ can be, and should be, designed to initialize themselves in some reasonable way, using a *constructor* function. Here is the length class rewritten with two constructor functions:

```
// File length.h
#define INCHES_PER_METER 39.3700787
class length {
  private:
    double d;
  public:
    length();
    length(double x);
    double cm();
    void cm(double x);
    double inches();
    void inches(double x);
};
```

Constructor functions are very special: they have the same name as the class, they have no return type, not even `void`, and they cannot be called directly, that is, `D.length()` is not allowed. When a variable of type length is first created, a constructor function runs automatically. The constructor `length()`, with no parameters, is called the default constructor. A reasonable default constructor would be:

```
length::length() {
  d = 0;
}
```

If for some reason you prefer to initialize to something other than zero, that's fine. Now when the main function first starts up, the declaration `length D` will automatically invoke the constructor and make sure that `d` is zero. Suppose you normally want to have `d` initialized to zero, but occasionally to something else? This is the job of the other constructor, which would look like this:

```
length::length(double x) {
  d = x;
}
```

Then we could do something like this:

```
int main () {
  length D(1.5);
  double x = 0;
  char choice = 'x';
```

Now when the variable `D` is first created, `d` will be set to 1.5. In this particular case this is probably a bad idea, because of the question of units. The "1.5" must be in meters, since that is how `d` is stored, but it's not obvious that it should be meters. Because of this, a better approach would be to have only the default constructor, and to require an explicit statement `D.cm(1500)` or perhaps `D.meters(1.5)` to set the value.

The bottom line is that you should always write a default constructor function, and add others as they seem to be useful and appropriate.

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;
}
```

5

More About Classes

We can do many useful operations with simple variables, things like `x+y`, `n++`, `cout << n`, `n=m`, and so on. It is very convenient to be able to do similar things with class objects, though of course some things may not make sense or have different meanings.

Consider strings, and again let's suppose that we need to write our own string class. We can easily imagine useful meanings for many operations; suppose `s` and `t` are string objects.

`cout << s`	Print the string on the screen.
`s=t`	Make `s` a copy of `t`.
`s+t`	Concatenate the two strings into one string.
`s+=t`	Concatenate `s` and `t` and save the result in `s`, that is, concatenate `t` onto the end of `s`.

C++ can't know what such operations should mean for every possible class we might design, so we must tell C++ what to do in those cases that we want to implement. When we extend the meaning of an operation like `+` to some new class, we are *overloading* the operator.

5.1 Overloading assignment

Perhaps the most fundamental operation is assignment, `s=t`. When we define a new class, C++ does have some idea of what this might mean, and is willing to perform `s=t` in a default way. Unfortunately, in many, perhaps most, cases this will not work. A typical class contains a number of variables of various types. C++ by default will perform `s=t` by copying each of the variables from `t` to the corresponding variable in `s`. In our string class, for example, C++ would essentially do this:

```
s.c = t.c;
s.lnth = t.lnth;
s.capacity = t.capacity;
```

This is almost certainly not what we want to do. Remember that `c` is a pointer to an array of characters. When we do `s.c=t.c` we make `s.c` point to exactly the same array that `t.c` points to; we do not make a copy. Then if the program changes the string `t`, say by adding some new characters, the same thing will happen to `s`. Even worse, it won't happen correctly, because `t.lnth` will be correctly modified to reflect the new length, but `s.lnth` will not.

So we have to intervene and overload the assignment operator. Here's how. First we modify the class definition:

```
class mystring {
private:
  char* c;
  int lnth, capacity;
public:
  mystring();
  mystring& operator=(const mystring& rhs);
  bool append(char x);
};
```

We are going to write a new function, `operator=`; this special syntax tells C++ that we are going to overload `=`. The return type is a mystring, which we know is how `=` normally works; we have added a new feature, putting `&` after the mystring. Recall that adding the `&` to a function parameter means that the function will not make a copy. Here it means the same thing: when the function returns a value, it will be the actual object we return, not a copy (normally, returning a value does make a copy). Likewise, we use `&` for the parameter `rhs` also, and add the label `const`; this tells C++ that we do not plan to modify `rhs`. Here is the function:

```
mystring& mystring::operator=(const mystring& rhs) {
  if ( this != &rhs ) {
    delete [] c;
    lnth = rhs.lnth;
    capacity = rhs.capacity;
    c = new char[capacity];
    for (int i=0; i<lnth; i++)
      c[i] = rhs.c[i];
  }
  return (*this);
}
```

There is a lot going on here. The variable `this` is always a pointer to the current object; it is automatically supplied by C++. The syntax `&rhs` means "the address of `rhs`," that is, it is a pointer to `rhs`. If `this == &rhs` it means that the two strings are literally the same object, because they have the same address. In this case, there is no point in doing anything, and in fact the very next line will destroy the string if `this` and `&rhs` are the same. If they are not the same, we start by deleting the current character array, so that we can copy the new character array in from `rhs`. We copy `lnth` and `capacity` directly, then we make a new `c` array, and copy the characters one at a time from `rhs.c` to `c`. Finally, we want to return the current object. We can't say `return(this)`, since that would return a pointer rather than the object itself. The new syntax `*this` means the object to which `this` points; this is called *dereferencing the pointer.*

Let's return to the function declaration:

```
mystring& operator=(const mystring& rhs);
```

We know that the use of `&` in both places prevents C++ from making a copy, but why did we choose to use it? *Because C++ doesn't know how to make a copy of a mystring object; this is precisely why we are overloading assignment to begin with.* Once we have done this, you might expect that if we want to write a function like

```
void convert(mystring s) { ...
```

we would be fine, because now C++ knows how to copy a string into the internal variable `s`. Unfortunately, this still won't work. When we use a function like `convert(t)`, C++ uses a different mechanism to copy `t` into `s`, called a *copy constructor.*

5.2 The Copy Constructor

Recall that a constructor runs when an object is first created. In the case of a function, when the function starts to run, any parameter variables are created, and C++ uses a constructor that copies an entire object into the new variable. But as with the assignment operator, C++ doesn't typically know how to do this, so we have to tell it. Fortunately, the copy constructor typically does almost exactly the same thing that the assignment operator does, so we don't have to rewrite it. First, we add a new constructor to the class:

```
class mystring {
private:
  char* c;
  int lnth, capacity;
public:
  mystring();
  mystring(const mystring& s);
  mystring& operator=(const mystring& rhs);
  bool append(char x);
};
```

and then we write the simple code:

```
mystring::mystring(const mystring& s) {
  c=NULL;
  (*this) = s;
}
```

First we set `c` to the special value `NULL`; without this, `c` probably contains some random, illegal address. Then in the assignment function, `delete [] c` will fail, because `c` is invalid. The delete function will recognize the value `NULL` as legal, but also as a "non-address", and so it will do nothing. Then we use the already defined assignment function to copy `s` into the current object, referred to as before with `*this`.

Now the convert function above will work fine, and in addition we can use the new constructor like any other constructor. For example, if `t` is a string, and we want a new string `s` to start as a copy of `t`, we can declare `s` like this:

```
mystring s(t);
```

If you are writing a class that will be used by others, or that has the potential to be used in other programs, you should overload the assignment operator and write a copy constructor.

5.3 More Overloading

Suppose we want to be able to use `s+t` to concatenate strings `s` and `t` into a single string. This should not change either `s` or `t`, so it makes sense to write the overloading function outside of the class. While we could do this directly, it is neater to notice that `s+=t` would also be convenient; this would concatenate `t` onto the end of `s`, changing `s` but not `t`. If we implement `+=` it becomes very easy to implement plain `+`.

Even though the `+` operator will not be inside the class, it is useful only in conjunction with the class, so we start by adding its declaration to `mystring.h`. We also add the `+=` operator inside the class.

```
class mystring {
private:
  char* c;
  int lnth, capacity;
public:
  mystring();
  mystring(const mystring& s);
  mystring& operator=(const mystring& rhs);
  mystring& operator+=(const mystring& rhs);
  bool append(char x);
};
mystring operator+(const mystring& s, const mystring& t);
```

The `+=` operator is very like the assignment operator, so we copy the form. In the `+` function we have used `&` parameters, though they are not required. We make the function more efficient by not requiring that the parameter values be copied.

Now in `mystring.cc` we put the definition of the `+` operator.

```
mystring operator+(const mystring& s, const mystring& t) {
  mystring temp(s);
  temp += t;
  return temp;
}
```

Note how simple this is: we copy `s` into a new string `temp` using the copy constructor; we concatenate `t` onto the end; we return the new string. Of course, we need to define `+=` to make this work; it is not much more complicated, and is much like the assignment operator:

```
mystring& mystring::operator+=(const mystring& rhs) {
  if ( this != &rhs ) {
    for (int i=0; i<rhs.lnth; i++) {
      append(rhs.c[i]);
    }
  } else {
    mystring temp = rhs;
    for (int i=0; i<temp.lnth; i++) {
      append(temp.c[i]);
    }
  }
  return (*this);
}
```

Again, note that this is quite simple because we have taken advantage of the already written `append` function, which not only adds a new character but increases the array size if necessary.

Next we see how to overload the input and output operators, `>>` and `<<`. These also are external to the class, but should be bundled in the same files. To `mystring.h` we add the lines

```
ostream& operator << (ostream& ostr, const mystring& rhs);
istream& operator >> (istream& istr, mystring& rhs);
```

Now the output operator is

```
ostream& operator << (ostream& ostr, const mystring& rhs) {
  for (int i=0; i<rhs.lnth; i++) {
    ostr << rhs.c[i];
  }
  return ostr;
}
```

Again we use pass by reference for the `rhs` parameter for efficiency; for the `ostream` parameter and return, we must use the reference `&` so that we use the real stream object both in and out. The actual code is quite simple, but there is a problem: `lnth` and `c` are private variables, so the function does not have access to them. We can fix this by making the function a `friend` of the `mystring` class; friends have the same access to private variables and functions that class objects do. We modify the class definition in `mystring.h` as follows:

```
class mystring {
friend ostream& operator << (ostream& ostr, const mystring& rhs);
private:
  char* c;
  int lnth, capacity;
public:
...
```

The normal behavior of `>>` on strings of characters is to skip "whitespace" and read characters up to the next whitespace. This makes this operator a bit trickier than the output operator.

```
istream& operator >> (istream& istr, mystring& rhs) {
  char nc;
  mystring temp;
  while ( (nc=istr.peek()) && (nc == ' ' || nc == '\n' || nc == '\t')) {
    istr.get();
  }
  while ( (nc=istr.peek()) && nc != ' ' && nc != '\n' && nc != '\t') {
    istr.get();
    temp.append(nc);
  }
  rhs = temp;
  istr.clear();
  return istr;
}
```

Here we make `rhs` a pass by reference parameter because we are going to modify it. We start by skipping spaces, end of lines, and tabs. The `peek` function returns the next character in the stream but does not "consume" it; once we know it is a whitespace character, we consume it with the `get` function. When the loop finds the first non-whitespace character, it leaves it in the stream for the next while-loop. The second loop consumes non-whitespace characters and uses the public function `append` to add them to the string. Note that because we are using `peek` again, the first whitespace character will be left in the stream, as it should be. The `clear` function sets the input stream into a good state in case the final read resulted in an error because there were no more characters.

This code reads the new characters into a temporary `mystring` variable, then uses `=` to overwrite `rhs`. We could have used `rhs` directly, with the result that the new characters would have been appended to the end of the existing `rhs` string. In other contexts the `>>` operator overwrites existing information, so that is the best policy here.

6

Template Classes

Some classes may naturally be thought of as "container" classes—classes that hold and manipulate some other type of data. Consider the following class:

```
class data_set {
private:
  int *D;
  int last,capacity;
  void merge (int, int);
  void mergesort (int,int);
public:
  data_set ();
  ~data_set ();
  data_set (data_set &);
  data_set& operator=(const data_set&);
  void add(int);
  bool get(int,int&);
  int size();
  void sort ();
  int find (int);
};
```

This class holds a "data set" consisting of integers, and provides some simple functions to manipulate the data. It is easy to imagine wanting a data set that will hold something other than integer values, like doubles, or strings, or personnel records. A data set is much like an array, but with some useful extra functions built in. Unlike an array, however, this class is limited to holding integer values. It's certainly not difficult to rewrite it to hold doubles or strings, but each would be a separate class. This is inconvenient compared to arrays: we can get an array of anything just by declaring it.

C++ in fact does allow us to define a sort of generic data set class, which we can turn into a real class to hold other types of data on demand. Instead of writing a whole new data set class every time we need a new one, we can write a **template** for such a class, then simply tell C++ to use the template to construct a data set to hold whatever type of data we want.

Starting at the end of the process, what we will do is see how to write a template data set class so that new data sets can be obtained with syntax like this:

```
data_set<int> L1;
data_set<double> L2;
data_set<string> dictionary;
```

These declarations create three data sets, one to hold ints, one to hold doubles, and one to hold strings. The basic idea for templates is simple: in the data set code, we use a placeholder name, say `base_type`, instead of an actual type like `int`. Then when the compiler sees a declaration like `data_set<int>`, it runs through the template replacing `base_type` by `int`, and compiles the resulting code to produce a class capable of maintaining data sets of integers. The syntax of templates takes some getting used to, but it is not intrinsically difficult. Figure 6.0.1 shows a "templatized" data set declaration. To illustrate more fully the syntax, the class has been expanded with some new functions, and we have added some external functions.

Note that before *each* new item (the class definition and the external operator functions) the special bit of code `template <class base_type>` is required. These definitions would be contained in a `.h` file as usual, say `data_set.h`. Note well that within the class definition the name of the class is simply `data_set`, while outside the class we must use `data_set<base_type>`.

The templatized code would go in `data_set.cc`, portions of which are shown in figure 6.0.2. In this code, any place we would have used, say, `int` (were we writing a data set to hold ints), we instead use `base_type`, and we replace the class name `data_set` with `data_set<base_type>`, *except* in the names of the constructor and destructor functions, where this is not allowed. It is permissible to leave out the `<base_type>` inside a class member function, but it is also permissible to include it.

Since `data_set.cc` contains merely templates for real functions, it does not make sense to compile `data_set.cc`. Instead, `data_set.cc` is treated much like a `.h` file. First we include it in any code file that needs it, using `#include ‘‘data_set.cc’’`, then when the compiler sees something like `data_set<double>` it can replace `base_type` by `double` everywhere and compile the resulting code.

```
template <class base_type>
class data_set {
private:
  base_type *D;
  int last,capacity,trace;
  void merge (int, int);
  void mergesort (int,int);
public:
  data_set ();
  ~data_set ();
  data_set (const data_set &);
  data_set& operator=(const data_set&);
  data_set& operator+=(const data_set&);
  void add(base_type);
  bool get(int, base_type&) const;
  int size () const;
  void sort ();
  int find (base_type) const;
};

template <class base_type>
data_set<base_type> operator+
   (const data_set<base_type>&, const data_set<base_type>&);

template <class base_type>
ostream& operator<<(ostream&, const data_set<base_type> &);

template <class base_type>
istream& operator>>(istream& istr, data_set<base_type> & d);
```

Figure 6.0.1 A data set template class

In reality, it's not quite so simple because most programs are written in multiple files, compiled separately, and then linked together. If two different files use data sets of integers, then we don't want two identical copies of the code to be generated and linked into the final executable program. There are two ways to deal with this: modify the linker to identify multiple copies of the code and put only one copy in the final program, or somehow guarantee that only one copy exists. There are two disadvantages to the first method: time is wasted compiling the code multiple times, and the linker has to be rewritten. The second method also has a disadvantage, namely, we need to guarantee that there is only one copy to begin with. Here again there are two options: make the compiler smart enough to create only one copy, or require the programmer to do it. The g++ compiler is capable of all three variations. I will describe here the simplest, though it requires some extra work

```
template <class base_type>
data_set<base_type>::data_set () {
  last = -1;
  capacity = 0;
  D = 0;
  trace = 0;
}

template <class base_type>
data_set<base_type>& data_set<base_type>::operator=
        (const data_set<base_type>& rhs) {
  if ( this != &rhs ) {
    delete [] D;
    last = rhs.last;
    capacity = rhs.capacity;
    D = new base_type[capacity];
    trace = rhs.trace;
    for (int i=0; i<=last; i++)
      D[i] = rhs.D[i];
  }
  return (*this);
}

template <class base_type>
ostream& operator<<(ostream& ostr, const data_set<base_type> & d) {
  base_type v;
  int i;
  for ( i=0; d.get(i,v); i++ ) {
    ostr << v << endl;
  }
  return ostr;
}
```

Figure 6.0.2 data set template code

by the programmer, namely, it is up to the programmer to provide a single copy of every required class.

For each data set class that you need, you create one small `.cc` file to instantiate that type, for example, you might use a file called `double_data_set.cc` to create a data set class for doubles. In other `.cc` files, as usual, you include only `data_set.h`, but in `double_data_set.cc` you include both `data_set.h` and `data_set.cc` (or include only `data_set.cc` if it in turn includes `data_set.h`). The rest of `double_data_set.cc` is then exceptionally simple, as shown in figure 6.0.3. These few lines tell the compiler to generate actual code. In other `.cc` files, use `data_set<double>` to refer to data sets of doubles; do not use the “template” keyword.

```
#include <fstream>
#include <iostream>
using namespace std;
#include "data_set.h"
#include "data_set.cc"

template class data_set<double>;

template
ostream& operator<<(ostream& ostr, const data_set<double> & d);

template
istream& operator>>(istream& istr, data_set<double> & d);

template
data_set<double> operator+(const data_set<double>& lhs,
                           const data_set<double>& rhs);
```

Figure 6.0.3 Creating a data set of doubles.

7

Inheritance

Sometimes a collection of classes have much in common, or perhaps an existing class has most but not all the functionality you want in a class. In such circumstances it is possible to use existing classes to build new classes, called subclasses, that start with the existing classes and then add or modify features as needed. As an example, I'll present part of a simple collection of classes to implement railroad cars.

We start with a very general class called a container, shown in figure 7.0.1. We expect the container to end up with a volume function and a reader function to report how full the container is. We want subclasses to be able to rewrite the reader function so we declare it to be "virtual". We also expect the volume function to be provided by subclasses, and there is no reasonable default, so we make it a "pure virtual" function by adding the "`=0`" at the end of the line. This means we do not need to provide the function as part of the container class.

We then write a second, still quite primitive, class as a subclass of the container class; see figure 7.0.2. This class will provide an actual volume function, and adds new dimension variables to the generic container.

Next we start the actual railroad car hierarchy of classes, with the base class shown in figure 7.0.3. The only thing common to all railroad cars will be a "year built" variable and a function for computing the age of the car.

Finally, we begin to introduce actual car types. A box car has the properties of both a box and a railroad car, and can inherit both of them simultaneously as shown in figure 7.0.4. It inherits the correct volume and age functions; I've added the constructor functions and a writer function to set the variables in an existing box car.

```
class container {
protected:
  double percent_filled;
public:
  container ();
  virtual double get_percent_filled ();
  virtual double volume () = 0;
};
```

Figure 7.0.1 Container Class

```
class box : public container {
protected:
  double height, width, length;
public:
  box ();
  box ( double l, double w, double h );
  virtual double volume ();
};
```

Figure 7.0.2 Box Class

```
class railroad_car {
protected:
  int year_built;
public:
  railroad_car();
  railroad_car(int yb);
  virtual int age();
};
```

Figure 7.0.3 Railroad Car Class

```
class box_car : public railroad_car, public box {
public:
  box_car();
  box_car(double l, double w, double h, double pct, int yb);
  void set (double l, double w, double h, double pct, int yb);
};
```

Figure 7.0.4 Box Car Class

Note that none of the classes have a private section, but the base classes have a new protected section. Any variables and functions in the protected section are not accessible to the "outside world," just as if they were in a private section, but protected variables and functions are available to subclasses. Private variables and functions are not available even to subclasses. In general, most variables should remain in private sections so that even subclasses must go through the standard public interface.

Once the classes are declared, the implementation of the functions proceeds almost as usual, as shown in figure 7.0.5. One thing that you can do, but not in a way that you're likely to guess, is to use a specific constructor from a base class. The default box car constructor uses the set function which sets all the variables directly; this is possible because those variables are protected, not private. The other box car constructor shows how you could use the box and railroad car constructors to set some of the variables. This would not work with the "percent filled" variable because a box car is not a direct descendant of a container. It would be possible to write another box constructor that accepts the percentage filled as a parameter. The advantage to calling on existing constructor functions is that this method will work when the variables in question are private.

```
box_car::box_car () { set (10,5,20,0,1900); }
box_car::box_car (double l, double w, double h, double pct, int yb)
  : box (l,w,h), railroad_car (yb) {
  percent_filled = pct;
}
void box_car::set(double l, double w, double h, double pct, int yb) {
  height = h; width = w; length = l;
  percent_filled = pct;  year_built = yb;
}
```

Figure 7.0.5 Some functions

You will have noticed that the word "public" is used when a subclass is declared to inherit the base class; not surprisingly, this "public" can also be "protected" or "private". The inheritance method controls the access further subclasses have to variables. If subclass B is declared to be a protected subclass, like this:

```
class B : protected A
```

then all public variables in class A become protected variables in class B. This means they are not available to the "outside world", but would be available to subclasses of class B. If instead we use

```
class B : private A
```

then public and protected variables from A become private in B, and are not available even to subclasses of B.

Another advantage to using subclasses is that a variable of a subclass type is automatically considered to be a variable of the base type as well. For example, if a function is declared as "`void fn(railroad_car R)`" and `B` is a box car, then you are allowed to call `fn(B)`. Unfortunately this will not work properly as written. When the compiler sees the definition of `fn` it arranges for a parameter of a certain size to be provided, a variable just big enough to hold a railroad car. A box car is bigger, and so when the value of the box car is copied into the variable `R` it won't work properly. To fix this, make the parameter a pass-by-reference parameter: "`void fn(railroad_car& R)`".

Similarly, if you declare "`railroad_car train[100]`" you get an array of 100 railroad car variables. It is legal to say "`train[i]=B`" if `B` is a box car, but again the box car value won't actually fit in the allotted space. To fix this problem, you must declare the array to contain pointers: "`railroad_car* train[100]`". Now you could say something like this:

```
box_car* B;
B = new box_car;
train[i] = B;
```

To access the box car at location `i` you would use something like "`(*train[i]).volume()`". This is done so often that C++ has a nice shortcut for this: "`train[i]->volume()`". The arrow symbol "`->`" (a hyphen and a greater-than symbol) takes the place of the `*` and the period. This is much easier to type and to read once you get used to it.

If an array can hold pointers to different types of railroad cars, or the railroad car parameter in a function can turn out to be any type of railroad car, it will be necessary to be able to tell what sort of car you have. One easy way to deal with this is to add a variable to the railroad car class to identify it; all constructors for actual cars should then set the car type correctly.

As usual, real understanding of inheritance requires that you do some programming. A good exercise would be to complete the definitions of the classes we've looked at, and to write new classes "cylinder" and "tank car" in addition. A better design would use private variables and more constructors and reader and writer functions, as indicated in figure 7.0.6.

```
#define BOXCAR  1
#define TANKCAR 2
class container {
private:
  double percent_filled;
public:
  container ();
  container (double pct);
  virtual void set_percent_filled (double pct);
  virtual double get_percent_filled ();
  virtual double volume () = 0;
};
class box : public container {
private:
  double height, width, length;
public:
  box ();
  box ( double l, double w, double h, double pct );
  void set_dimens (double l, double w, double h);
  virtual double volume ();
};
class cylinder : public container {
};
class railroad_car {
private:
  int year_built;
  int car_type;
public:
  railroad_car();
  railroad_car(int ct);        // ct is car type
  void set_year_built (int yb);
  int get_year_built ();
  void set_car_type(int ct);
  int get_car_type();
  virtual int age();
};
class box_car : public railroad_car, public box {
public:
  box_car();
  box_car(double l, double w, double h, double pct, int yb);
  void set (double l, double w, double h, double pct, int yb);
};
class tank_car : public railroad_car, public cylinder {
};
```

Figure 7.0.6 A better starting point.

8

Linked Lists

In computer science, a "list" is pretty much what you would guess: some data items arranged in a one-dimensional sequence in a particular order. Lists are used all the time, so it's not surprising that they're used in computer programs as well. One way to implement lists in a computer program is by using an array, but this is not the only way to do it. Arrays have some very attractive properties: they are relatively easy for the compiler to implement, they are efficient in practice, they allow any item in the array to be accessed directly (you do not need to access the first hundred items before you can access the hundred and first item). But arrays are not well suited to all tasks. For example, if some data is stored in sorted order in an array, it is relatively difficult to insert a new item somewhere in the middle while maintaining order—lots of data has to be moved to create the free spot for the new item.

Arrays work by assigning each spot in the array to a predetermined memory location that is easy to find. Another way to maintain a list of items is to have each item contain information pointing to the next item on the list; any list organized in this way is called a "linked list". This means that adjacent items on the list need not be adjacent in memory, and it allows new items to be inserted anywhere without moving existing data around. On the other hand, it becomes much more difficult to find the hundred and first item on the list, since now the only way to find it is to start at the beginning and follow the pointers through the first hundred items.

Neither of these methods is "perfect"—in some applications the advantages of arrays outweigh the disadvantages, and in others linked lists are preferable. It is important to understand how to implement linked lists for the latter occasions.

There are in fact many different ways to implement linked lists, depending on where the links are stored and what sort of information they hold. We'll look here at one method in detail: the links are stored in class objects separate from the data, and the links themselves are C pointers. A schematic view of this scheme is shown in figure 8.0.1.

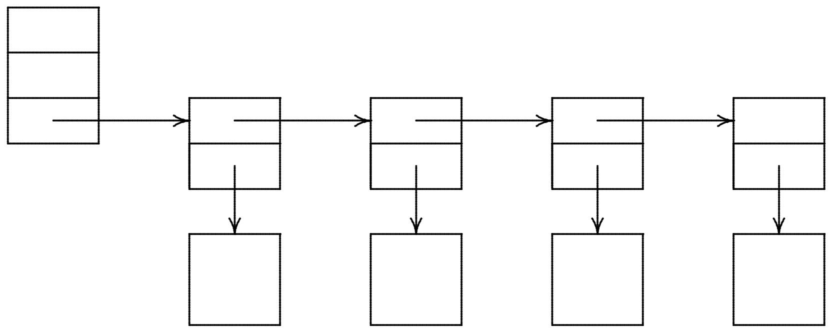

Figure 8.0.1 A linked list

The links are shown as square boxes divided into two sections; the data items are the undivided square boxes; the three-section box is a header for the list; it is how the list is found and referred to elsewhere in the program.

Each link except the last has a pointer to the next link, and each link points to one data item—the data item that is at that position on the list. The header contains a pointer to the first link and hence allows us to find the first data item on the list. Depending on the demands that will be placed on the linked list, the link and header classes may have other member variables. If the linked list is designed for general purpose use, as a tool for other programmers, the available variables and functions should generally be more inclusive, to make sure that the list will be as useful as possible.

In the version pictured it is quite easy to go from one item to the next on the list, but it is not easy to go to the previous item; this can be fixed by adding a pointer from each link to the previous link. There is no way from a link to discover what list the link belongs to; this can be remedied by adding a pointer from each link to the header.

The header contains a pointer to the first link on the list. It is also often useful to have a pointer to the last link on the list. Since it is time consuming to discover how big a list is, a variable in the header that records the number of items on the list is desirable. We will of course need to make sure that whenever an item is added to or removed from the list this counter is updated. One way to organize this is to make sure that only the header class is allowed to add or remove items. The header will require the ability to keep track of a position in the list at which new items are to be added or removed. A simple way to arrange this is to add another pointer to the header that at all times points to the "current" link. Whenever an item is added or removed, the change happens at the current location. The header will of course need to be able to adjust the position of current. With all of this in mind, the class definitions in figure 8.0.2 should be mostly self-explanatory. The header class is actually named list, so that in use a list can be declared as something like `list<student_record> L`.

```
template <class base_type>
class listlink;

template <class base_type>
class list {
 private:
  listlink<base_type> *start, *last, *current;
  int the_size;
 public:
  list();
  ~list();
  int size();
  bool is_empty();
  void reset();
  bool add_after(base_type* n);
  bool add_before(base_type* n);
  bool add_first(base_type* n);
  bool add_last(base_type* n);
  base_type* remove();
  bool advance();
  bool backup();
  bool go_first();
  bool go_last();
  base_type * access();
};

template <class base_type>
class listlink {
  friend class list<base_type>;
 private:
  listlink *next, *prev;
  base_type * data;
  list<base_type> * head;
  listlink();
  listlink(base_type *n);
};
```

Figure 8.0.2 Classes for linked lists

There is one new piece of the C++ language evident here: "`friend class list`". We want to make sure that only the header alters the list. By putting everything in the link class in the private section, we guarantee that the outside world can't mess with the link, but then the header also can't access or modify the link data. By declaring class list to be a "friend" of class listlink we allow the header to have access to all private data and functions in the link class.

Let's look at an implementation of a few of the functions, beginning with `add_after` in figure 8.0.3. We begin by creating a new link and pointing it at the data item. Then we link the new item in after the current link, unless `current` is the null pointer, in which case we put the new item at the beginning of the list.

```
template <class base_type>
bool list<base_type>::add_after(base_type* n) {
  listlink<base_type> * l = new listlink<base_type>(n);
  if (current) {
    if (current->next) {
      current->next->prev = l;
    }
    l->next = current->next;
    l->prev = current;
    current->next = l;
    if (last==current) last=l;
  } else {
    l->next = start;
    l->prev = NULL;
    if (start) start->prev = l;
    else last = l;
    start = l;
  }
  l->head = this;
  the_size++;
  return true;
}
```

Figure 8.0.3 The `add_after` function

Removing a data item is a bit trickier than adding one, as illustrated in figure 8.0.4. It is not safe to delete the data item itself, as it may still be in use. On the other hand, if the link contains the last pointer to the data item, simply deleting the link results in a memory leak. We want to delete the link and return a pointer to the data item, so that the caller can decide what to do about the data item. We also need to decide what to do with `current` after we remove the current item. The code shown moves current to the link following the removed link, if there is one, and moves it to the link preceding the removed link otherwise, unless of course the current link is the only link, in which case `current` becomes the null pointer.

The `advance` function is simple but does have a few potential pitfalls. If `current` is the null pointer, we advance it to the first item on the list, except of course if the list is

```
template <class base_type>
base_type* list<base_type>::remove() {
  if (!current) return NULL;
  listlink<base_type>* new_current;
  base_type* d = current->data;
  if (current == start) {
    start = current->next;
    if (start) start->prev = NULL;
    else last = NULL;
    new_current = start;
  } else {
    current->prev->next = current->next;
    if (current->next) {
      current->next->prev = current->prev;
      new_current = current->next;
    } else {
      last = current->prev;
      new_current = last;
    }
  }
  delete current;
  current = new_current;
  the_size--;
  return d;
}
```

Figure 8.0.4 The `remove` function

```
template <class base_type>
bool list<base_type>::advance() {
  if ( !current )
    if ( !start ) return false;
    else {
      current = start;
      return true;
    }
  if ( !current->next ) return false;
  current = current->next;
  return true;
}
```

Figure 8.0.5 The `advance` function

empty. Otherwise `current` really points to a link, so then we check its `next` field to see if we can advance. We return `true` if we are able to advance `current` and `false` otherwise.

It is frequently desirable to examine the items on a list without disturbing the `current` pointer. It might even be necessary to keep track of more than one additional location on the list. On the other hand, we probably don't want to allow any modifications to the list except through the header and its `current` pointer. The standard approach here is to define a third "iterator" class that contains its own version of `current` and is allowed to examine data items, but is not allowed to add or remove data. The class definition is shown in figure 8.0.6. Our iterator class needs to access the data in the header and the links, so we need to add "`friend class listiterator`" to both the `listlink` and `list` classes. The variables and functions of the iterator class will be almost a subset of the list class, and the function definitions will be almost identical. We add one constructor and a head variable so that an iterator can be associated with a particular list; indeed, in our implementation, the default constructor will produce a useless iterator since it will not be associated with any list. A writer function could be added to allow an existing iterator to be associated with a new header.

```
template <class base_type>
class listiterator {
 private:
  listlink<base_type> *current;
  list<base_type> * head;
 public:
  listiterator();
  listiterator(list<base_type>* head);
  void reset();
  bool advance();
  bool backup();
  base_type * access();
};
```

Figure 8.0.6 The iterator class