Some More C++

B. J. MacLennan, 2009-12-01

C++ is a huge language, and we have covered only its basics in CS 102. Every class and every textbook must make choices about what is most important to cover, and I have followed the lead of our textbooks, *How to Think Like a Computer Scientist in C++* and *Learning Computing with Robots in C++*. Nevertheless, a few of the C++ topics we have not covered are common enough that later classes may assume you know them. You can learn them easily enough from other C++ books and online resources, but I have prepared this document to explain some of them. If you cannot understand them from these brief explanations, then you will have to consult a C++ reference, but at least you will know they exist and what they are called.

1. C-arrays

C++ has inherited many features from C, a much older (and more primitive) language. One of these is the *array*, which is much like a vector, but its size is fixed. The simplest way to declare an array is to put its size in brackets after the its name. For example,

```
double hours[7];
```

declares hours to be an array with elements indexed hours[0] to hours[6], just like a vector. An array can be given an initial value as follows:

```
double hours[7] = {0.0, 8.0, 4.0, 10.0, 8.0, 8.0, 0.0};
```

If you want the array to be just big enough to hold the initial values, then you can omit the size and let the compiler do the counting for you:

```
double hours[] = {0.0, 8.0, 4.0, 10.0, 8.0, 8.0, 0.0};
```

You can declare two-dimensional arrays (or matrices), which are a lot like vectors of vectors. For example,

```
int mat[2][3];
```

is an array, whose elements are indexed mat[0][0], mat[0][1], mat[0][2], mat[1][0], mat[1][1], mat[1][2]. This is also the order in which the elements are stored in memory (called *row-major order*). A two-dimensional array can be initialized like this:

```
int mat[2][3] = {{1, 2, 4}, {1, 3, 9}};
```

You can let the compiler count the number of rows:

```
int mat2[][3] = {{1, 2, 4}, {1, 3, 9}, {1, 4, 16}, {1, 5, 25}};
```

C++ allows you to declare arrays of three or more dimensions in a similar way. The result is the same as mat[4][3].

Vectors have a size() member function, but there is nothing analogous for C-arrays, so you have to know how long it is, preferably by giving a name to its size, e.g.,

```
const int size_hours = 7;
double hours [size_hours];
```

In particular, when you pass a C-array to a function, the function cannot use size() to control loops etc., so it must know the length of its argument array, or the length must be passed as an additional parameter.

When you pass an array to a function (e.g., f(hours)), it does not copy the array; rather, it passes the address where the array begins in memory (effectively passing it by reference). The formal parameter corresponding to the array can be declared in any of the following ways:

```
void f (double A[7]);
void f (double A[]);
void f (double* A);
```

The first suggests (but does not enforce) the fact that f() expects only 7-element arrays; the second suggests the argument could be any length. The third corresponds to the actual implementation, i.e., a pointer (address) is being passed. Regardless of how the formal parameter is declared, the elements of the array can be accessed using either indexing (e.g., A[i]) or through pointer arithmetic (e.g., *(A+i); see #4 below).

2. C-strings

(a) Definition

C has no built-in string data type, and so when we speak of *C-strings* we mean arrays of characters. For example, char str[100]; declares str to be a array that can hold strings with up to 100 character. Since C-arrays are of fixed size, so are C-strings, which would be inconvenient for many purposes. For example, we might want str to hold any string of length up to 100. Since the string stored in a character array can be shorter than than the size of the char array, C and C++ mark the end of the actual string with a special *null character*, written '\0'. To initialize the C-string msg to "Bye." you could write:

```
char msg[5] = {'B', 'y', 'e', '.', '\0'};
```

This is rather ugly, so C++ allows the initialization to be abbreviated:

```
char msg[5] = "Bye.";
```

Notice that you have to reserve space for the null character even though you don't have to write it. Indeed, whenever you see a double-quoted string of characters in C++, it is really an abbreviation for a null-terminated array of characters. An array of C-strings is really a two-dimensional char array, e.g.,

```
char month[12][10] = {"January", "February", "March", …,
  "September", "October", "November", "December"};
```

Notice that we need 10 columns to accommodate the null character at the end of "September" (the longest month name). (We could have omitted "12" had we wanted.)

(b) <cstring> Library

The <cstring> library provides a number of useful functions for operating on null-terminated C-strings. I'll mention a few of them here; see a C++ reference for more information. The function strlen(S) returns the length of string S. The function strcmp(S,T) compares two strings; if they are equal it

returns 0, if S is lexicographically greater than T it returns a positive number, and if S is lexicographically less than T, it returns a negative number. The invocation strcpy(S,T) copies string T into string S, but you must ensure that S is big enough to contain T, or it will overrun the storage after S. (Note that S = T will not work on C-strings.) Finally, strcat(S,T) concatenates T onto the end of S (analogous to S += T), but you must ensure that S is big enough to hold the concatenated strings.

3. Formatted Input with scanf

In *LCR* ch. 7 (pp. 184–9) you learned about formatted output using the printf function in the <cstdio> library. There is also a formatted input function, scanf, which I'll describe briefly; see the online documentation for more. The invocation scanf(format, &v1, …, &vn) attempts to read n values into the variables v1, …, vn according to format, which is a C-string. The most common format specifiers are %d to read a decimal integer, %lf to read a double, %c to read a character, and %s to read a C-string. The invocation returns an int, which is the number of values scanf() successfully read. If it is less than n, then the input did not fit the provided format (e.g., you tried to read an integer and there were alphabetic characters). If scanf() returns the special value EOF (often equal to -1), then an eof was encountered. Here is a simple example:

```
int data;
if (scanf("%d", &data) != 1)
  { cout << "Bad input!\n; exit(1) }
```

If you don't care about errors, then you can use scanf() as a statement (see #7 below), e.g.,

```
double rate;
scanf("%lf", &rate);
```

4. Pointer Arithmetic

C++ permits a limited amount of arithmetic on pointers, but more than enough to get you into trouble! To get the idea, look at the following code:

```
double hours[7];
double* pd;
pd = hours;
```

The pointer pd has been declared to be a pointer to a double variable, and the assignment pd = hours causes it to point to the first element of hours, that is, it contains the address of hours[0]. The expression pd+1 is also a pointer, namely to the next value in memory, which in this case will be hours[1]. To see why, suppose that hours begins at address 708644; after the assignment, that value will be in pd. Further suppose, as is often the case, that doubles occupy 4 bytes. Then, since pd has type double*, pd+1 will be the address 708648 (== 708644 + 4), which is the location of hours[1].Thus you can fetch the i-th element of hours with either hours[i] or *(pd+i). Also, if pd points to hours[0], then pd++ will advance it to hours[1], and in this way you can increment your way through an array (which on many computers is *very very slightly* faster than using indexing). Similarly, you can do pd-1 and pd--, which also illustrates a problem. If pd points to hours[0], then pd-1 points to hours[-1], which does not make sense, since it is not in the array hours at all! It is the address

708640, which might not even contain a double number; it could be an int, four characters, or anything else, but your program will treat it like a double and get meaningless results. In other words, with pointer arithmetic you can point to just about any location in your memory area, but that doesn't mean you will get meaningful results. This can create hard-to-diagnose program bugs.

You have seen that you can either add an integer to a pointer or subtract an integer from a pointer to get another pointer of the same type. You can also subtract two pointers of the same base type, which gives the integer number of base-type values between the two pointers. This makes sense if both pointers are pointing into the same array, but doesn't make much sense in most other cases. You can compare pointers for equality and inequality (==, !=), which is useful. You can also compare them with the order operators (<, >), but that doesn't make much sense unless they happen to be pointers into the same array.

5. Command-line Arguments

As you know, many programs that are invoked from the linux command line can take one or more *command-line arguments* (e.g., vi myfile.cpp or cp file1 file2). Programs that you write can also take command-line arguments, and it's not difficult to use them. Replace your usual "int main ()" function header with the following:

```
int main (int argc, char* argv[])
```

The first parameter, argc (argument count), is the number of command line arguments provided to your program. The second parameter, argv (argument value), is a C-array of pointers to C-strings. This is not as complicated as it sounds; argv[1] is a C-string containing the first argument, argv[2] is the second, and so forth. Also, argv[0] is your program's name (the "zeroth argument"). For example, if your compiled program is in a file called myprog, and the user writes on the command line:

```
myprog infile.dat outfile.txt
```

then argc will be 3, argv[0] will be "myprog", argv[1] will be "infile.dat", and argv[2] will be "outfile.txt". Note that argc is always at least 1. You can use the functions in <cstring> to manipulate the arguments (see #2).

6. Type Definition

C++ allows you to give a new name to a data type. For example,

```
typedef double real;
```

makes real a synonym for double. After this definition, you can use real in variable declarations, function headers, etc. There are two main reasons for doing this. One is that it allows you to give a more meaningful, application-oriented name (such as real) to a machine data type (such as double), thus improving the readability of your program. Another reason is that it makes your programs more maintainable, since if you later decide to represent real numbers by a different machine type, such as float or long double, you only have to change it in one place.

7. Expressions as Statements

We normally use *expression* to refer to code that returns a value (such as N*5 or 0<x) and *statement* to refer to code that does something besides compute a value (e.g., N=10, return, if, for). However, in C++ any expression can be used as a statement; its value is simply thrown away. For example, if you write

```
N*5;
```

in your program, it will compute N*5, then throw the result away and go on to the next statement. This feature is pretty useless unless the expression has some side-effect (such as changing memory or doing I/O) in addition to computing a value. Then you might use the expression as a statement if you are more interested in the side effect than in the value computed. One very common example is to use the N++ and N-- for their side-effect (incrementing/decrementing N), while ignoring their value (discussed in #8). You might also call a non-void function as a statement if you were interested in what the function does but not the value it returns. An example is scanf() (see #3).

8. Pre/Post Increment/Decrement

In addition to the operators N++ and N--, which you know, there are also operators ++N and --N, which have the same effect on N (incrementing or decrementing it, respectively), but differ in the values they return. The difference is that ++N increments N and its value is the value of N after it has been incremented, whereas N++ increments N, but its value is the value of N before it was incremented. You can tell which is which, because the ++ is before N if you increment before you fetch N's value, and it follows N if you fetch N's value before you increment it. These are called *pre-increment* and *post-increment*, respectively. Clear? The same applies to the *pre-* and *post-decrement* operators --N and N--.

Of course, if you restrict yourself to using these as statements, rather than expressions (see #7), then there is no difference between the pre- and post- forms. With regard to the use of N++ as an expression the author of *ThinkCS* says (Sec. 7.10), "I am not going to tell you what the result is," since he feels the notation is too confusing. It's a point worth considering; you don't have to use a feature just because it's there. (However, you will, no doubt, see it in other people's programs.)

9. Nonzero Integers as Boolean true

As you remember, the bool value false is equivalent to the integer 0 and the bool value true is equivalent to 1. (It's as though there were a built-in declaration enum bool {false, true};.) In point of fact, in conditional expressions (in if, while, for, etc.) any nonzero integer value is interpreted as true. This encourages some "expert" programmers to write less-than-transparent code. For example, they might write

```
for (N=100; N; N--) { … }
```

instead of the clearer

```
for (N=100; N>0; N--) { … }.
```

In the old days, the former was slightly more efficient, but with modern compilers there is no difference. Nowadays, readability and transparency of purpose are much more important.

10. General for-Loops

Recall that a for-loop, "for (*initializer*; *condition*; *updater*) { *body* }" is equivalent to

```
initializer
while (condition) {
   body
   updater
}
```

The *initializer* and *updater* can be multiple statements (or expressions, see #7 above) separated by commas. Here is an example:

```
for (i=0, p=1; i <= n; i++, p *= x)
  cout << x << "^" << i << " = " << p << endl;
```

Any of the parts can be omitted if they are not needed. If the body would be empty, it can be replaced by a semicolon:

```
for (n = 0, r = x; r /= 2; n++);
```

Can you figure out what this loop does? (Recall #9.) Although the C++ for-loop is very powerful and flexible, it is also an invitation to write opaque, difficult-to-maintain code.

11. Conditional Operator

You are familiar with the C++ conditional *statement*, "if(—) — else — ", but C++ also has a conditional *expression*, which is sometimes convenient. The expression "C ? T : F" has the value T if conditional C is true, and the value F otherwise. For example, the conditional statement

```
if (A > B) max = A;
else max = B;
```

can be written more concisely as a conditional expression:

```
max = (A > B) ? A : B;
```

12. malloc() and free()

These are functions for allocating and deallocating memory, which C++ has inherited from C. They are ancestors of C++'s new and delete, which are safer to use. The invocation malloc(N) allocates a block of N bytes from the heap and returns a pointer to it. You have to know how many bytes to allocate for the type of data you intend to put in the block, but a built-in function sizeof() will help you figure it out. Also, malloc() returns a void* pointer, so you will have to cast it to another kind of pointer to use it in your program. For example, the following two declarations are effectively equivalent:

```
double* ptr = (double*) malloc (sizeof(double));
double* ptr = new double;
```

The effect of free(ptr) is the same as delete ptr. Both malloc() and free() (as well as the related functions calloc() and realloc()) are in the <cstdlib> library. Avoid using new/delete and malloc()/free() is the same program, because they are not guaranteed to play well together.

13. Preprocessor Directives

C++ has many preprocessor directives, which are used to modify your source code before it is processed by the compiler. You are familiar with the #include directive, which inserts a header file or other C++ text into your program. Here I will mention a couple more that are useful.

(a) #define

The #define preprocessor directive is used to define a *macro*, which is an abbreviation for some arbitrary program text. For example,

```
#define ERROR cout << "You have made error: " <<
```

makes ERROR an abbreviation for "cout << "You have made error: " <<". The macro text begins with the first non-blank character ("c" in this case) and continues up to the end of the line. Thereafter, if you code

```
ERROR "Bad Input\n";
```

it will be just as though you wrote

```
cout << "You have made error: " << "Bad Input\n";
```

It is the custom in C++ programming to give macros names in all uppercase letters so that they can be distinguished from other names. You are familiar with the standard macro NULL.

Consider the following macro definition:

```
#define UTILS
```

This defines UTILS to be the empty macro; that is, UTILS will be replaced by the empty string. This might seem pointless, but it has an important use, which I'll discuss now.

(b) #ifdef and #ifndef in Header Files

The C++ preprocessor can handle conditional statements similar to C++ if-statements. Here I will mention only two variants. The directive #ifdef means "if defined" and the directive #ifndef means "if not defined." Suppose you have the following in your program:

```
#ifdef MYMAC
…
#endif
```

If the macro MYMAC is defined, then everything from the #ifdef to the #endif will be included in your program, otherwise that text will be skipped as though it wasn't there. #ifndef MYMAC is similar, but opposite: if MYMAC is not defined, then the text up to #endif will be included, but if it's defined, the text will be skipped. This can be used to solve a common problem in program organization.

Suppose you have a header file utils.h for a bunch of utility functions that you find useful, so you often #include it in your source code. You also have a header file graphics.h for some graphics classes that you have implemented, which also use (and #include) utils.h. Now the problem is that if your main program includes utils.h (because it needs some utilities) and also includes graphics.h (because it does some graphics), then it will end up including utils.h twice, which will lead to duplicate declaration errors. There are ways to avoid this, but they are unwieldy for large programs with many included files. The simple solution is to wrap the utility declarations in utils.h with the following directives:

```
// file utils.h
#ifndef UTILS
#define UTILS
// declarations of the utilities
 …
#endif
```

The first time utils.h is included, UTILS will not be defined, so it will process the text up to the #endif. This will #define UTILS and include all the utility declarations. If utils.h is included again, UTILS will be defined already, and so #ifndef UTILS will skip the #define and everything else up to the #endif (i.e., all the declarations in utils.h).

~~~~~

In conclusion, there is much more to the C++ features you learned about in class as well as to those described here, and there are many other C++ features that I have not mentioned at all. If you continue programming in C++ you can learn about them from other C++ books and from online reference material.
~~~~~