Computation

Chapter 4

Computation

All a program ever does is take some input and produces output.

The input can come from a keyboard, mouse, touch screen, files, other input devices, other programs and other parts of the program

To deal with data a program contains some sort of data structure or state.

Output

Just as input can come from all over, output can go to various locations.

Screen, files, other output devices, other programs and other parts of the same program.

Inputs to a part of a program are called *arguments* and output is called *results.*

Computation definition

By *computation* we simply mean the act of producing some outputs based on some inputs.

For example producing the result 49 from the input 7 using the computation square.

Side note, before the 50’s a computer was defined to be a person who did computations, such as an accountant, a navigator, or a physicist.

Objectives and tools

Our job as programmers is to express computations:

1. correctly
2. simply
3. efficiently

The order is important.

It doesn’t matter how fast something is if it’s wrong.

Quality code

Concerns for structure and “quality of code” are often the fastest way of getting something to work correctly.

When done well, the need to debug is minimized.

Good program structure during development can minimize the number of mistakes made and the time needed to search for and fix errors.

Main tools

The main tools for organizing a program are

1. Abstraction
2. Divide and conquer

Abstraction

Abstraction: is to hide details that we don’t need to use a facility behind a convenient and general interface.

We don’t have to know how cin works with >> in order to use it, we simply need to know that it will work.

Divide and conquer: is to break larger problems down into several little ones.

But why?

A program built out of parts will be larger than one that is written in one part.

We simply cannot write and maintain large monolithic programs.

You don’t get good code from simply writing a lot of statements.

Expressions

The most basic building block of programs is an expression.

C++ will evaluate expression starting at the top of the file and continue to the bottom.

The simplest of expressions are literal values: 10, ‘a’, 3.14, “Norah”

Names of variables are also expressions:

```
int length = 20;
int width = 40;
int area = length * width;
```

More complicated examples

We can combine expressions using operators and if needed grouping them with parenthesis.

```
int perimeter = (length+width)*2;
```

or

```
int perimeter = length*2+width*2;
```

The basic precedence rules you know from math apply here.

Use parenthesis to help group terms.

A note about readability

Why should you care about readability.

You will most likely be reading your code.
Others maybe reading your code, e.g. me.

Ugly code is not only hard to read, it is also much harder to get correct.

Ugly code often hides logical errors.

Constant expressions

A constant expression is one that cannot be changed.

```
const double pi = 3.14159;
pi = 7; //error
double c = 2 * pi; //looks good
```

Non-obvious literals in code are called magic constants and should be avoided.

Operators

The operators were discussed in chapter 3 and are more or less what you expect.

+, -, *, /

Note that a < b < c does not do what you might expected.

a < b is a boolean value so you need to join that with && (logical and) or || (logical or)

a < b && b < c

Conversions

We can mix our integers and doubles in our expressions.

But what will C++ do?

Basically, if C++ “sees” a double on the right hand side of an expression, it will convert all other integers to doubles to do the calculation.

If everything is an integer, then integer math is used.

If everything is a double, then floating point math is used.

But what about the left hand side

So if we have

```
int x  = 7 * 9;
```

we get the int result 63.

```
int x = 7.0 * 9.0;
```

The right hand side will compute the result 63.0 then it will look at the type on the left hand side.

It will truncate the result and store 63 in x.

Better

```
int x = int(7.0) * int(9.0);
```

Statements

An expression computes a value from a set of operands using operators.

What do we do when we want to produce several results, repeat an expression or choose among alternatives?

We use a statement.

A statement is just a language construct to allow us to structure our program.

Kinds of statements

So far we have seen expression statements and declaration statements.

An expression is simply a statement followed by a semicolon.

a = b;

++b;

What is a = b ++ b; //a syntax error

Do we mean a = b++; b; or a = b; ++b;

Semicolons end our statements.

Empty statement

There is a statement that does nothing, the empty statement.

```
;
if ( x == 5 ) ;
{ y = 3; }
```

This is probably not what we mean, but it will compile and run.

It is a very common error.

Selection statements

When we have to choose among a set of alternatives, we have two choices:

1. if statement
2. switch statement

if statements

The simplest form of a selection is an if-statement which selects between two alternatives.

```
if ( a< b )
    cout << “max(” << a << “,” << b << “) is ” <<
b << “\n”;
else
    cout << “max(” << a << “,” << b << “) is ” <<
a << “\n”;
```

if-statements again

An if-statement chooses between two alternatives.

If its condition is true, the first statement is executed; otherwise the second statement is.

Sample

```
int main()
{
   const double cm_per_inch = 2.54;
   double length = 1;
   char  unit = 0;
   cout << “Please enter a length followed by a unit (c or 
i):\n”;
   cin >> length >> unit;
   if ( unit == ‘i’ )
      cout << length << “in == ” << cm_per_inch * length 
<< “cm\n”;
   else
      cout << length << “cm == ” << length / cm_per_inch 
<< “in\n”;
   return 0;
}
```

How good was that?

What was left out?

Better

```
int main()
{
    const double cm_per_inch = 2.54;
    double length = 1;
    char  unit = 0;
    cout << “Please enter a length followed by a unit (c or 
i):\n”;
    cin >> length >> unit;
    if ( unit == ‘i’ )
        cout << length << “in == ” << cm_per_inch * length 
<< “cm\n”;
    else if ( unit == ‘c’ )
        cout << length << “cm == ” << length / cm_per_inch 
<< “in\n”;
    else
        cout << “Sorry, I don’t know a unit called ” << unit 
<< “\n”;
    return 0;
}
```

Details

if (expression) statement else statement

The pattern is,
the keyword *if*,
followed by an *expression* in parenthesis,
followed by a *statement*,
followed by an *else*,
followed by a *statement*.

if-else if-else...

We can build complex set of if-else statements.

The second else can be followed by another if-else statement.

switch-statements

The switch-statement is an alternative to the if-statements

There are some technicalities

1. The value on which we switch must be of an integer, char, or enum type.
2. The values of the case labels must be constant expressions.
3. You cannot use the same value for two case labels.
4. You can use several case labels for a single case.
5. Don’t forget to end each case with a break.

```
int main()
{
    const double cm_per_inch = 2.54;
    double length = 1;
    char  unit = 0;
    cout << “Please enter a length followed by a unit (c or i):\n”;
    cin >> length >> unit;
    switch (unit){
    case ‘i’:
        cout << length << “in == ” << cm_per_inch * length <<
“cm\n”;
    break;
    case ‘c’:
        cout << length << “cm == ” << length / cm_per_inch <<
    “in\n”;
    break;
    default:
        cout << “Sorry, I don’t know a unit called ” << unit <<
    “\n”;
    break;
    }
}
```

Missing the break

If you leave off the break statements the compiler will accept it.

The matching case, will “drop” through to the next case, causing both cases to execute.

Iteration - the while-statements

```
int main()
{
    int i=0;
    while ( i<100 )
    {
        cout << i << ‘\t’ << square(i) << ‘\n’;
        ++i;
    }
}
```

What we need for loops

1. A way to repeat some statements (to *loop*)
2. A variable to keep track of how many times we have been through the loop (a *loop variable* or a *control variable*), here the int called i.
3. An initializer for the loop, here 0;
4. A termination criterion, here, that we want to go through the loop 100 times.
5. Something to do each time the loop (the *body* of the loop)

Blocks

Note how we grouped the two statements the while had to execute.

A sequence of statements delimited by curly braces, { and }, is called a *block*.

A block is a special kind of statement.

The empty block is sometimes useful for expressing that nothing is to be done.

```
if ( a <= b ) { }
else {
    int t = a; a = b; b = t; }
```

for-statements

Iterating over a sequence of numbers is very common in C++.

A for-statements is purpose built for this problem.

The difference is the management of the control variable is done at the top.

```
for ( int i=0; i<100; i++ )
    cout << i << '\t' << square(i) << '\n';
```

A for-statement is always equivalent to a while-statement.

Functions

In the previous example, what was square(i)?

It’s a call of a function.

A *function* is a named sequence of statements.

A function can return a result (also called a return value).

We don’t have to use the result of a function call but we do have to give a function exactly the arguments it requires.

Example

```
square(2); //probably an error, unused return
int v1 = square(); //error missing argument
int v2 = square; //error, parenthesis missing
int v3 = square(1,2); //error too many
arguments
int v4 = square(“two”); //error wrong type
```

Function body

The *function body* is the block that actually does the work.

```
{
    return x*x;
}
```

Function definition syntax

type identifier (parameter-list) function-body

The *type* is the return type.

The *identifier* is the function’s name.

The *parameter-list* is a list of arguments separated by commas. The list can be empty.

The *function-body* is the block of statements to be executed.

If the function does not return a value, then *void* is used as the return.

Example

```
void write_sorry()
{
    cout << “Sorry\n”;
}
int square( int x )
{
    return x*x;
}
```

But why?

Writing functions:

1. makes the computations logically separate.
2. makes the program text clearer (by naming the computation).
3. makes it possible to use the function in more than one place in our program.
4. eases testing.

Programs are usually easier to write and to understand if each function performs a single logical action.

Function declarations

In order call a function, all we need to do is this:

int x = square(44);

So all we need is the first line: int square(int x);

Most programs don’t want to look at a function body.

The first line with a semicolon is called the *function declaration*