2 Testing and Looping

2.1 Conditions 19
2.2 Conditional Expressions 22
2.3 Loops 23
2.4 Example: Computing the Frame 24
2.5 Counting 27
2.6 Loop Design 29
2.7 Assertions 34
2.8 Order Notation 37
2.9 Example: Maximum Subsequence 39

The control flow of a program is determined by tests and loops. In this section, we review the main C++ structures for branching and looping and discuss some aspects of loop design. In particular, we look at how to systematically write ***correct*** and ***efficient*** loops.

Engineering is the application of theory (mathematics, physics, etc.) to practice. Programming has not yet reached the maturity of engineering because we do not yet have adequate theories. We can prove that trivial programs satisfy a specification, but the techniques do not scale well.

Nevertheless, we should do what we can. It is feasible, for example, to use systematic techniques, rather than guesswork, to code loops, and these techniques are introduced in Section 2.3 on page 23.

2.1 Conditions

A ***condition*** is an expression whose value is either ***true*** or ***false***.[4]

C++ has a standard type, `bool`,[5] with exactly two values, `true` and `false`. Type `bool` also provides operators, as shown in Table 1. 27

Table 1: Boolean operators

Operation	Logic Symbol	C++ operator
conjunction	$\wedge$	`&&`
disjunction	$\vee$	`||`
negation	$\neg$	`!`

C++ also has operators that work on all of the individual bits of their operands in parallel, shown in Table 2. Do not confuse these with the boolean operators!

lazy evaluation

The Boolean operators `&&` and `||` are ***lazy***; this means that they do not evaluate their right

[4]This assumes a two-valued logic. There are other logics with more than two values. For example, the value of a formula in a three-valued logic might be ***true***, ***false***, or ***unknown*** (used, for example, in relational database systems).

[5]Named after George Boole, the English mathematician who wrote *The Laws of Thought*, where he describes what we now call *Boolean logic*, which has become a foundation of modern computer science.

Table 2: Bitwise operators

Operation	C++ operator
complement	~
shift left	`<<`
shift right	`>>`
and	`&`
exclusive or	`^`
inclusive or	|

operands unless they need to. In detail, `p && q` is evaluated like this:

1. Evaluate `p`. If it is `false`, yield `false`.
2. Otherwise, evaluate `q` and yield the result.

Similarly, `p || q` is evaluated like this:

1. Evaluate `p`. If it is `true`, yield `true`.
2. Otherwise, evaluate `q` and yield the result.

Lazy evaluation has both good and bad consequences:

- It is more efficient, because the right operand is not evaluated unnecessarily.
- 28 It enables us to write tests such as

```
if (y != 0 && x / y <= MAX_RATIO)
    ....
```

- In C++, conjunction (and) and disjunction (or) are not commutative! That is, `p && q` is not always equivalent to `q && p`. However, they cannot yield different truth values: at worst, one would succeed and the other would fail.

C++ also provides ***comparison operators***, which compare two operands and yield a boolean value. They are `<` (less than), `<=` (less than or equal to), `==` (equal to), `!=` (not equal to), `>` (greater than), and `>=` (greater than or equal to). Do not confuse `==` with `=` (the assignment operator). These work with operands of most types but may not always make sense.

How to read C++:

`x == y`	≡	"`x` equals `y`"
`a = b`	≡	"`a` gets `b`" or "`a` becomes `b`" or "`a` is assigned `b`"

In addition to the type `bool`, C++ inherits some baggage from C. In C, and therefore in C++:

- 0 is considered `false`
- Any non-zero value is considered `true`

This convention has several consequences:

- A condition such as `counter != 0` , in which `counter` is an integer, has the same truth-value as `counter`. Many C++ programmers therefore use the simpler form, `counter`, in a context where a truth value is expected.

- Expressions that are `true`, when considered as truth values, are not necessarily equal. For example, suppose we want to express the fact that two counters are either both zero or both non-zero. We could express this condition as

  ```
  (counter1 == 0) == (counter2 == 0)
  ```

  or, equivalently, as

  ```
  (counter1 != 0) == (counter2 != 0)
  ```

  Using the abbreviation above, we might be tempted to simplify this to

  ```
  counter1 == counter2
  ```

  But this expression has a different meaning!

Write conditions carefully. Prefer complete expressions to abbreviations.

The convention also suggests the question: what ***is*** zero in C++? The answer is that there are many things that are considered to be zero — and are therefore also considered to be `false` in a condition:

- Integers of type `short`, `int`, and `long`, with value 0
- Floating point numbers of type `float`, `double`, and `long double`, with value 0.0
- The character `'\0'`
- Any null pointer (we will discuss pointers later)
- The first element of an enumeration
- The `bool` value `false`
- And perhaps others . . . .

The empty string is ***not*** equivalent to `false`:

```
    const string emptyString = "";
```

2.2 Conditional Expressions

A ***conditional expression*** is an expression with a boolean value. Conditional expressions are often called ***tests***, for short.

29 Programs typically evaluate conditions using `if` statements, which have one of two forms:

```
if ( ⟨condition⟩ )
   ⟨statement⟩
```

or

```
if ( ⟨condition⟩ )
   ⟨statement⟩
else
   ⟨statement⟩
```

and work as you would expect them to.

The ⟨*statement*⟩s in an `if` statement can be simple or compound. This example illustrates both
30 possibilities:

```
if (angle < PI)
{
   cout << "Still going round ...";
   angle += 0.01;
}
else
{
   angle = 0;
}
```

The braces around the final assignment, `angle = 0`, are not essential. We could alternatively have written

```
....
else
   angle = 0;
```

Whether you include the braces or not is a matter of taste and preference. Including them adds two lines to the code (or one line if you put the left brace on the same line as `else`). But including them makes it easy to add another line later – which is often necessary – and avoids the error of adding a line but forgetting to add the braces.

The `?:` Operator. C++ also knows the ternary ***conditional operator*** `?:` (in fact, it is the only operator that requires three operands): ?: operator 31

⟨*condition*⟩ `?` ⟨*expresssion1*⟩ `:` ⟨*expression2*⟩

It works like this: if the ⟨*condition*⟩ evaluates to `true`, the result is ⟨*expression1*⟩, otherwise ⟨*expression2*⟩. For example, to assign the bigger of two variables `a` and `b` to a variable `x`, you could write

```
x = a > b ? a : b;
```

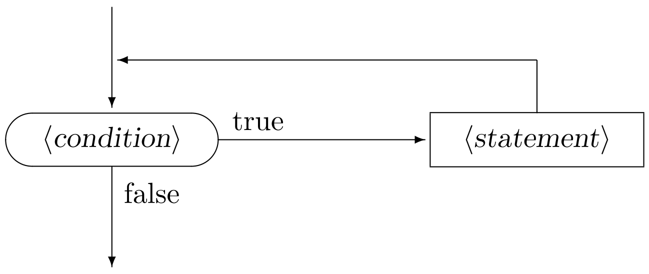
2.3 Loops

C++ provides three looping constructs. In the order in which you should consider using them, they are: `for`; `while`; and `do`/`while`.

The `while` loop has this structure 32

```
while ( ⟨condition⟩ )
   ⟨statement⟩
```

and it works like this:



An important feature of the `while` loop is that the loop body may not be executed at all. This is a very useful feature and, when writing a `while` loop, you should always check that its behaviour when the condition is initially `false` is correct.

A loop of this form 33

```
⟨initialize⟩
while ( ⟨condition⟩ )
{
   ⟨action⟩
   ⟨step⟩
}
```

should usually be written more concisely in this (almost) equivalent form:

```
for ( ⟨initialize⟩ ; ⟨condition⟩ ; ⟨step⟩ )
{
   ⟨action⟩
}
```

34 For example, instead of writing

```
int i = 0;
while ( i < MAX )
{
   doSomething(i);
   ++i;
}
```

write this:

```
for (int i = 0; i < MAX; ++i)
{
   doSomething(i);
}
```

in C++, prefer '++i' to 'i++'

It is a good idea to get into the habit of using pre-increment (`++i`) rather than post-increment (`i++`). For integers, it doesn't make much difference, but `++` and `--` are overloaded for other types for which the difference is more significant. The reason for preferring pre-increment is that `i++` may force the compiler to generate a temporary variable, whereas `++i` does not.

The `do`/`while` loop is used only when you want to evaluate the condition ***after*** performing the loop body: 35

```
do
   ⟨statement⟩
while ( ⟨condition⟩ )
```

2.4 Example: Computing the Frame

36 37 The program in Figure 7 on the next page uses `if`, `while`, and `for` statements. Figure 8 on page 26 shows an example of its use. The outer loop, formatting the rows of the display, is a `for` loop, because exactly one row is processed during each iteration. This pattern does not work for the inner loop, because progress is not always one column at a time. The `if` statements make decisions about what text to output, and they all have complex conditions with `&&` and `||` operators.

This program is easier to modify than Figure 6 on page 15: to change the size of the frame, all we have to do is change numbers in the `const` declarations. In fact, just changing the value of `pad` will change the spacing all around the greeting.

Design programs so that a few easily changed parameters change the behaviour of the program in a consistent way.

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
   cout << "Please enter your first name: ";
   string name;
   cin >> name;
   const string greeting = "Hello, " + name + "!";
   const int pad = 1;
   const int rows = pad * 2 + 3;
   const string::size_type cols = greeting.size() + pad * 2 + 2;
   cout << endl;
   for (int r = 0; r != rows; ++r)
   {
      string::size_type c = 0;
      while (c != cols)
      {
         if (r == pad + 1 && c == pad + 1)
         {
            // We are positioned for the greeting.
            cout << greeting;
            c += greeting.size();
         }
         else
         {
            if (r == 0 || r == rows - 1 ||
                c == 0 || c == cols - 1)
               // We are on a border.
               cout << "*";
            else
               cout << " ";
         ++c;
         }
      }
      cout << endl;
   }
   return 0;
}
```

Figure 7: Greeting.3

```
Please enter your first name: Wilberforce

***********************
*                     *
* Hello, Wilberforce! *
*                     *
***********************
```

Figure 8: A dialogue with Greeting.3

Source Code Comments. The original program (Koenig and Moo 2000, page 29) contains a
38 number of comments:

```
// say what standard-library names we use
using std::cin;         using std::endl;
....
// ask for the person's name
cout << "Please enter your first name: ";
....
// read the name
string name;
cin >> name;
```

These comments are acceptable, but only because this program appears in an introductory text book. In general, comments like this should not be written unless:

- They provide information that is not obvious from the code
- They make the code more readable without adding noise to it

There is one comment in this program which might serve a purpose:

```
// the number of blanks surrounding the greeting
const int pad = 1;
```

Without the comment, the meaning of `pad` would not be obvious, although it is not hard to guess its meaning by reading the next few lines of code. But any comment that is provided to explain the role of a variable raises an immediate question: could we eliminate the need for the comment by choosing a better name?

In this case, we could replace `pad` by `spaceAroundGreeting`, or some such name. Then the comment would be unnecessary.

Of course, it takes longer to type `spaceAroundGreeting` than `pad`. But the time programmers take to type a name a few times (five times for this program) is negligible compared to the time maintainers take to figure out what they meant.

39 Another problem with Figure 7 on the preceding page is the mysterious numbers 2 and 3:

```
const int rows = pad * 2 + 3;
const string::size_type cols = greeting.size() + pad * 2 + 2;
```

Although there is a comment explaining the meaning of `pad`, there is no comment explaining the formulas `pad * 2 + 3` and `pad * 2 + 2`. Again, we can excuse the authors in this case, because the explanation appears in the book (Koenig and Moo 2000, page 18–22). In production code, however, these numbers should be accompanied by explanatory comments or even defined as constants.

We could write the definitions in a way that shows how the values are obtained. This requires more typing but should not make any difference to the compiled code:

```
const int rows = 1 + pad + 1 + pad + 1;
const string::size_type cols = 1 + pad + greeting.size() + pad + 1;
```

Compiler Warnings. Why does the program above use `int` as the type of `rows` – which seems quite natural – and the curious expression `string::size_type` as the type of `cols`? This type is used because it is the type returned by the function `string::size()`. It is an integer type, but we don't know which one (probably `unsigned long` but possibly something else). If we declare "`const int cols`", the compiler issues a warning: 40

`string::size_type`

```
frame.cpp(15) : warning C4267: 'initializing' :
   conversion from 'size_t' to 'const int', possible loss of data
```

We don't want warning messages when we compile, and one way to get rid of this message is to use the correct type.

If the compiler issues warning messages, revise your code until they disappear.

It is not always easy to eliminate warnings, but the effort is worthwhile: Even if it doesn't save your time now, it may save a maintainer's time later.

2.5 Counting

In everyday life, if we have N objects and want to identify them by numbers, we assign the numbers $1, 2, 3, \ldots, N$ to them. Another way to look at this is to say that the set of numbers forms a ***closed interval*** which we can write as either $1 \le i \le N$ or $[1, N]$.

Programmers are different. Given N objects, they number them $0, 1, 2, \ldots, N - 1$. This set of numbers forms a ***semi-closed*** (or ***semi-open***) interval which we can write as either $0 \le i < N$ or $[0, N)$. The advantages of semi-closed intervals include:

- They reduce the risk of "off-by-one" or "fencepost" errors.

  fencepost error

  To see why off-by-one errors are called "fencepost errors", consider the length of a fence with N posts spaced 10 feet apart. The length of the fence is $10(N - 1)$ feet: see Figure 9. 41

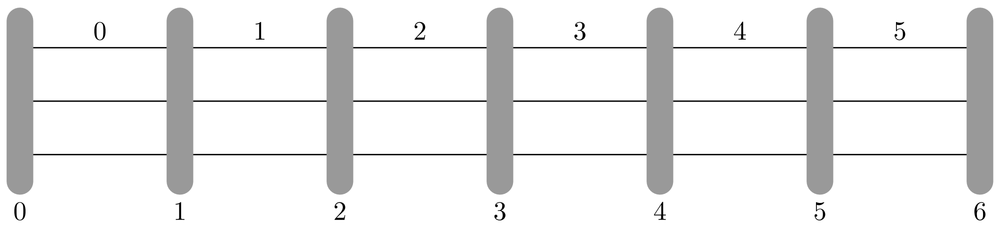


Figure 9: Fencepost numbering

- The closed interval $[M, N]$ has $N - M + 1$ elements; the semi-closed interval $[M, N)$ has $N - M$ elements, which is easier to remember and calculate.
- In particular, the closed interval $[M, N]$ is empty if $M > N$, which many people find counter-intuitive. The semi-closed interval $[M, N)$ is empty if $M = N$, which is easier to remember and check.
- Semi-closed intervals are easier to join together. Compare
$$[A, B),\ [B, C),\ [C, D), \ldots$$
to
$$[A, B - 1],\ [B, C - 1],\ [C, D - 1], \ldots$$
- The index of the first element of an array A in C++ is 0. The address of the I'th element of the array is $\&A + sI$ where $\&A$ is the address of the array and s is the size of one element (we will discuss array addressing in more detail later).

Typical C++ loops do ***not*** have the form

```
for (int i = M; i <= N; ++i) ....
```

in which `M` is often 1. Instead, they have the form

```
for (int i = M; i < N; ++i) ....
```

in which `M` is often zero. Note that, in the first case, `N` is the last element processed but, in the second case, `N` is the first element ***not*** processed. In fact, we will see later that there are good reasons for writing the termination condition as `!=` rather than `<`:

```
for (int i = M; i != N; ++i) ....
```

Start a range with the index of the first item to be processed; end the range with the index of the first item not processed. The first index of a range is often 0.

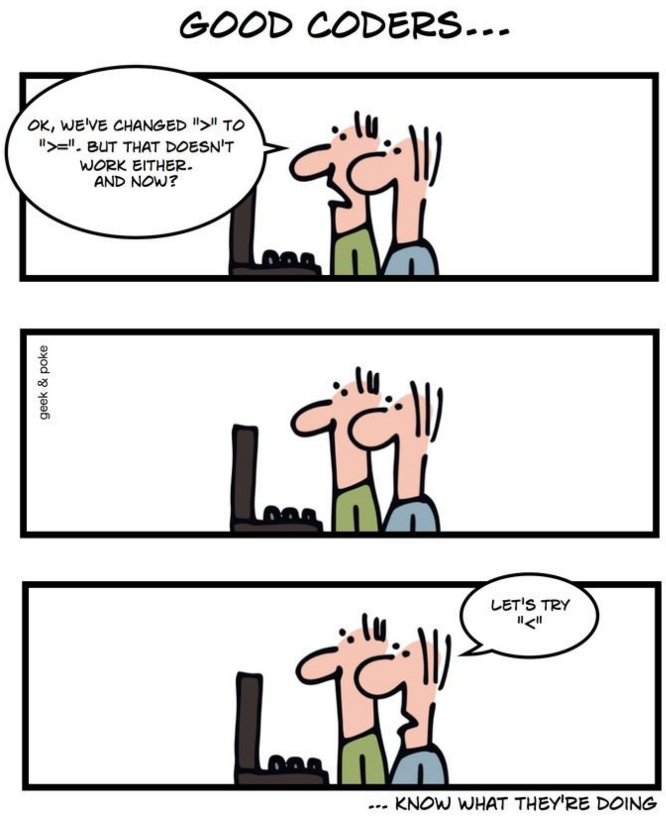

GOOD CODERS...
OK, WE'VE CHANGED ">" TO ">=". BUT THAT DOESN'T WORK EITHER. AND NOW?
geek & poke
LET'S TRY "<"
... KNOW WHAT THEY'RE DOING

Figure 10: There should be a better way…

2.6 Loop Design

Figure 10[6] shows a popular, but unreliable approach to loop design. Let's try to do better.

We discuss the design of loops with the assumption that they are going to be `while` loops. If they later turn out to have the appropriate pattern, we can convert them to `for` loops. With experience, we learn to recognize loops that have the `for`-loop pattern in advance and avoid the conversion step.

We design `while` loops using the following schema (the numbers are for reference, not part of the code): $\boxed{42}$

```
1    ⟨initialize⟩
2    //  $I$
3    while (C)
4    {
5        //  $I \wedge C$
6        ⟨body⟩
7        //  $I$
8    }
9    //  $I \wedge \neg C$
```

[6]Copyright Geek&Poke (http://geek-and-poke.com/), licensed under Creative Commons Attribution 3.0 Unported (CC BY 3.0) https://creativecommons.org/licenses/by/3.0/deed.en_US

loop invariant

- The comment on line 2 says that, after initialization, the condition I is true. I is the ***invariant*** of the loop.
- The comment on line 5 says that, at the start of the body of the loop, the loop invariant I and the loop condition C are both true. This provides an ***assumption*** that we can use in coding the loop.
- The comment on line 7 says that, after the body of the loop has been executed, the invariant I is still true (this is what being an ***invariant*** means).
- The comment on line 9 says that, when the loop exits, the invariant I is true but the loop condition C is false.

"Good programmers instinctively know what the invariant should be in a `while` ***loop."*** — ***Joel Spolsky,*** **Joel on Software,** ***page 164.***

2.6.1 Counting Lines

As a simple example of loop design, we consider the problem of writing `rows` lines of output, as in Figure 7 on page 25. We use a counter `r` to count the number of lines written and, following the convention for initializing counters, we will initialize it to zero. We will make the minor change of writing `ROWS` rather than `rows`, to indicate that `ROWS` is a constant and to improve readability.

Here is a suitable invariant: r *lines have been written*. Note that initializing r to zero makes the invariant true, because we haven't written any lines yet.

If we have printed $ROWS$ lines, there is no more to do. Consequently, the condition for the `while` loop is `r != ROWS` and the code begins: 43

```
1       int r = 0;
2       // r lines have been written
3       while (r != ROWS)
4       {
5           // r lines have been written and r != ROWS
```

The body of the loop must generate one line of output. We don't care (for this exercise) what that output will be, so we will just write a `cout` statement. The body must also count the lines produced. Thus the code continues:

```
6           cout << .... << endl;
6.1         // r+1 lines have been written
6.2         ++r;
7           // r lines have been written
8       }
9       // r lines have been written and not (r != ROWS)
```

Line 6 generates one line of output. This ***invalidates*** the invariant, as the comment on line 6.1 shows. Incrementing `r` makes the invariant valid again. We note that an invariant is not ***always*** true, but is true at certain well-defined points in the program. Also, whenever we perform an action that invalidates the invariant, we must perform another action (`++r` in this case) that makes it valid again.

Line 9 can be simplified as follows: 44

```
     r lines have been written and not (r != ROWS)
⇒   r lines have been written and r == ROWS
⇒   ROWS lines have been written
```

which is exactly what we needed.

Additional points:

- If we had written the `while` condition as `r < ROWS`, this reasoning would lead to a different conclusion:

  ```
       r lines have been written and not (r < ROWS)
  ⇒   r lines have been written and r >= ROWS
  ```

  That is, we could claim only that the code generates ***at least*** `ROWS` lines of output. This is correct, but it is less precise than the original conclusion, which is that the code generates ***exactly*** `ROWS` lines of output. One advantage of `!=` over `<` as a `while` condition is that it gives us a more precise conclusion. (This advantage was first pointed out by (Dijkstra 1976, page 56n). We will discuss other advantages later, in connection with the STL.)[7]

- This is an example of the situation mentioned above: the final code matches the pattern of the `for` statement and we can write the solution with a `for` loop. The invariant still applies: 45

  ```
  for (int r = 0; r != ROWS; ++r)
     // r lines have been written
     cout << .... endl;
  // ROWS lines have been written
  ```

- In addition to the invariant, which expresses something that does not change, we need something in the loop body that ***does*** change. Otherwise, the loop condition would never be satisfied and the loop would never terminate. In this example, the thing that changes is obviously the row counter; in other cases, it might not be so obvious.

- Suppose that we start counting from 1. A plausible invariant is: "`r` is the next line to be written", but this turns out not to be an invariant, because it is not true after we have written the last line. An invariant that works is "`r-1` lines have been written", which yields the following code in Figure 11 on the following page. The code is correct, but it is 46 more complicated and error-prone than the solution that counts from zero. As previously mentioned, the last line implies only that ***at least*** `ROWS` lines have been written rather than ***exactly*** `ROWS` lines have been written.

[7] However, this only applies to integer data types – never use equality/inequality comparisons with floating point numbers!

```
1       int r = 1;
2       // r-1 lines have been written
3       while (r <= ROWS)
4       {
5           // r-1 lines have been written and r <= ROWS
6           cout << .... << endl;
6.1         // r lines have been written
6.2         ++r;
7           // r-1 lines have been written
8       }
9       // r-1 lines have been written and not (r <= ROWS)
```

Figure 11: Starting from 1

Loop Coding Errors: The Zune Bug. Despite the fact (or, perhaps, especially because) loop conditions often seem deceptively simple, they nevertheless are one of the most common causes of bugs in software products. As an example, consider the code in Figure 12 that caused a crash[8] of the Zune media player on 31.12.2008.[9] It is part of a function that computes the number of years from the number of days that have passed since a certain starting date (in this case, 01.01.1980). As you can see, the code has to take leap years into account. Why did it crash on
$\boxed{47}$ 31.12.2008 (hint: 2008 was a leap year)?

```
while (days > 365)
{
    if (IsLeapYear(year))
    {
        if (days > 366)
        {
            days -= 366;
            year += 1;
        }
    }
    else
    {
        days -= 365;
        year += 1;
    }
}
```

Figure 12: The cause of the Zune crash 31.12.2008

[8]See http://www.engadget.com/2008/12/31/30gb-zunes-mysteriously-begin-to-fail-at-12am-december-31st/

[9]The Zune software is actually written in C#, not C++, but the lesson here is really about loop design.

2.6.2 Finding Roots by Bisection

Suppose that f is a continuous real-valued function, $A < B$, and $f(A) \leq 0$ and $f(B) > 0$. Then a fundamental theorem of real analysis says that there must be a value of x such that $f(x) = 0$ and $A \leq x < B$; that is, a ***root*** (or ***zero***) of f. Using the bisection method, we find a sequence of approximations to x by halving the interval $[A, B)$ yielding smaller intervals $[a, b)$. Figure 13 illustrates the process. $\boxed{48}$

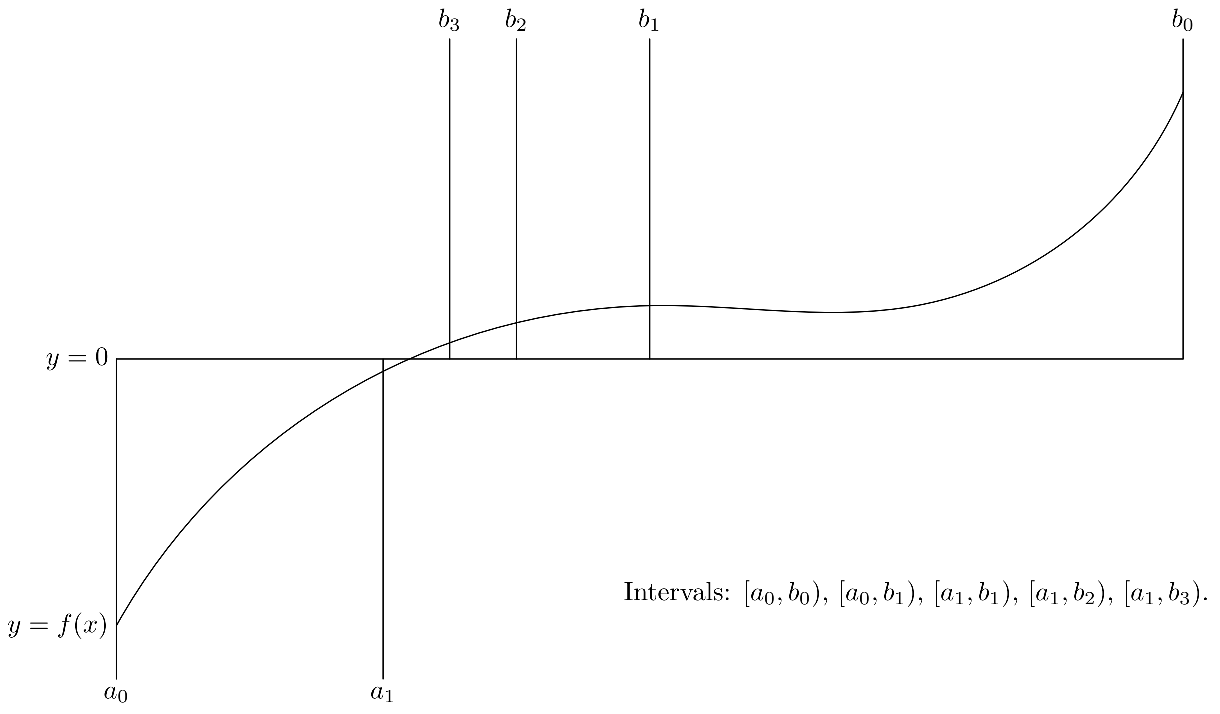

b_3
b_2
b_1
b_0
$y = 0$
Intervals: $[a_0, b_0)$, $[a_0, b_1)$, $[a_1, b_1)$, $[a_1, b_2)$, $[a_1, b_3)$.
$y = f(x)$
a_0
a_1

Figure 13: Finding zeroes by bisection

Part of the invariant is $f(a) \leq 0 \wedge f(b) > 0$. This ensures that there is a root of f in $[a, b)$. To be complete, we will also require $a < b$. We cannot expect to find the root exactly, so we will stop when the interval is sufficiently small; specifically, when $b - a < \varepsilon$. This suggests that the loop condition should be $\neg(b - a < \varepsilon)$, which is equivalent to $b - a \geq \varepsilon$. Thus we have: $\boxed{49}$

```
double a = A;
double b = B;
//  $I \equiv f(a) \leq 0 \wedge f(b) > 0 \wedge a < b$
while (b - a >= eps)

   ....

//  $I \wedge b - a < \varepsilon$
```

Note that the final comment, obtained by and'ing the invariant and the negation of the `while` condition, is what we need: there is a root within a small interval.

To complete the loop body, we find the midpoint of the interval $[a, b)$, which is at $m = (b - a)/2$. If $f(m) > 0$, there must be a root between a and m. If $f(m) \le 0$, there must be a root between m and b. We can write the code below. Note carefully how the `if` statement maintains the invariant. 50

```
double a = A;
double b = B;
// $I \equiv f(a) \le 0 \wedge f(b) > 0 \wedge a < b$
while (b - a >= eps)
{
   double m = 0.5 * (a + b);
   if (f(m) > 0)
      b = m;
   else
      a = m;
   // $I$
}
// $I \wedge b - a < \varepsilon$
```

51

52

Figure 14 on the next page expands this idea into a complete function and a test program. The `assert` statement is discussed in Section 2.7. When this program is run, it displays: 53

```
pi = 3.14159
e  = 2.71828
Assertion failed: a < b && f(a) <= 0 && f(b) > 0 &&
                  "bisect: precondition violation",
  file f:\courses\comp345\src\bisect\bisect.cpp, line 13
```

2.7 Assertions

54 The program in Figure 14 on the next page contains an ***assertion***:

```
assert(a < b && f(a) <= 0 && f(b) > 0 &&
      "bisect: precondition violation");
```

As with most programming constructs, there are three things that are useful to know about assertions: syntax *(what do we write?)*; semantics *(what happens?)*; and pragmatics *(when and why do we use assertions?)*.

```
#include <iostream>
#include <cassert>
#include <cmath>

using namespace std;

double bisect(
    double f(double),
    double a,
    double b,
    double eps = 1e-6 )
{
    if (a > b)
        return bisect(f, b, a, eps);

    assert(a < b && f(a) <= 0 && f(b) > 0 &&
          "bisect: precondition violation");

    while (b - a > eps)
    {
        // a < b && f(a) <= 0 && f(b) > 0
        double m = 0.5 * (a + b);
        if (f(m) > 0)
            b = m; // f(b) > 0
        else
            a = m; // f(a) <= 0
    }
    return 0.5 * (a + b);
}

double logm(double x)
{
    return log(x) - 1;
}

int main()
{
    cout << "pi = " << 0.5 * bisect(sin, 4, 8) << endl;
    cout << "e  = " << bisect(logm, 0.5, 3) << endl;
    cout << "e  = " << bisect(logm, 3, 3) << endl;
}
```

Figure 14: Finding zeroes by bisection

Syntax. Any code unit that uses assertions must contain either the (preferred) new-style directive

```
        #include <cassert>
```

or the old-style directive

```
        #include <assert.h>
```

assert The assert statement itself has the form

`assert(` $\langle$*condition*$\rangle$ `);`

Semantics.

- When the program executes, the $\langle$*condition*$\rangle$ is evaluated.
- If the $\langle$*condition*$\rangle$ yields `true`, or anything equivalent to `true`, the `assert` statement has ***no effect***.
- If the $\langle$*condition*$\rangle$ yields `false`, or anything equivalent to `false` (i.e., any kind of zero), the program is terminated with an error message.

The precise form of the error message depends on the compiler. In general, it will contain the text of the $\langle$*condition*$\rangle$, the name of the file in which the `assert` statement occurs, and the line number of the statement within the file.

Pragmatics. Here is a reliable guide to the use of assertions:

If an assertion fails, there is a logical error in the program.

Think of `assert(`C`)` as saying "I, the programmer, believe that C should always be true at this point in the program". Then the failure of an assertion implies that the programmer's belief was mistaken, which further implies that there was something wrong with the reasoning and therefore something wrong with the program.

55 Assertions are useful for expressing preconditions, postconditions, and invariants.

- A ***precondition*** is a condition that should be true on entry to a function. It imposes an obligation on the caller to ensure that the arguments passed to the function are appropriate. The assertion in Figure 14 on the preceding page is of this form.
- A ***postcondition*** is a condition that should be true when a function returns. It is a promise by the function to the caller that the function has done its job correctly.
- An ***invariant*** is a condition that should be true at certain, well-defined points in the program. For example, a loop invariant.

A useful trick for assertions is to append "`&&` ⟨*string*⟩" to the condition, in which ⟨*string*⟩ describes what has gone wrong. This does not change the value of the condition, because any string is considered to be non-zero and therefore `true`, but it may improve the diagnostic issued by the compiler.

For example, when the assertion 56

```
assert( a < b && f(a) <= 0 && f(b) > 0 &&
        "bisect: precondition violation");
```

in the program of Figure 14 on page 35, the run-time system displays

```
Assertion failed: a < b && f(a) <= 0 && f(b) > 0 &&
                "bisect: precondition violation",
file f:\courses\comp345\src\bisect\bisect.cpp, line 13
```

Assertions can be disabled by writing "`#define NDEBUG`" ***before*** "`#include <cassert>`". If the assertions were not failing, the only effect of this will be to save a few nanoseconds of execution time. Since conditions that were evaluated with `NDEBUG` undefined are no longer evaluated with `NDEBUG` defined, it is important that:

`#define NDEBUG`

Asserted conditions must not have side-effects.

Exceptions provide another way of recovering from errors; we will discuss them later in the course (Section 10.2 on page 203).

2.8 Order Notation

It is useful to have a concise way of describing how long a program or algorithm takes to run. The conventional way of doing this is to use "Big O" notation.

$\mathcal{O}()$ notation

> This section presents a simplified view of Big O notation. The Bachmann-Landau family of notations defines $\Omega(\cdot)$, $\Theta(\cdot)$, $o(\cdot)$, and $\omega(\cdot)$ as well as $\mathcal{O}(\cdot)$.

The time taken to compute something usually depends on the size of the input. We will use n to stand, in a general way, for this size. For example, n might be the number of characters to be read, or the number of nodes of a graph to be processed. If the time required is ***independent*** of n, we say that the ***time complexity*** is $\mathcal{O}(1)$. If the time required increases linearly with n, we say that the complexity is $\mathcal{O}(n)$.

Formally, $\mathcal{O}(\cdot)$ defines a ***set*** of functions: 57

$$g(n) \quad \in \quad \mathcal{O}(f(n))$$

if and only if there are constants A and M such that, for all $N > M$, $g(N) < A \cdot f(N)$.

Big-oh notation does two things: it singles out the dominant term of a complicated expression, and it ignores constant factors. For example,

$$\begin{array}{rrcl} & n^2 & \in & \mathcal{O}(n^2) \\ \text{and} & 1000000n^2 & \in & \mathcal{O}(n^2) \end{array}$$

because we can chose $A = 1000001$ in the definition above. Also

$$n^3 + 100000n^2 + 100000n + 100000 \quad \in \quad \mathcal{O}(n^3)$$

because, for large enough n, the first term dominates the others. By similar reasoning

$$n + 10^{-10}n^2 \quad \in \quad \mathcal{O}(n^2)$$

even though the second term looks very small.

Informally, we don't say "is a member of $\mathcal{O}(n^2)$" but, less precisely, "the complexity is $\mathcal{O}(n^2)$" (or whatever the complexity actually is). 58

Figure 15 on the next page shows a small part of the ***complexity hierarchy***. Each line defines a set of functions that is a proper subset of the set defined on the next line. So, for example, $\mathcal{O}(n) \subset \mathcal{O}(n \log n)$ (all linear functions are log-linear), and so on.

Very few algorithms are $\mathcal{O}(1)$. We use this set to describe operations that have a constant time bound. For example, "reading a character" is (or at least should be) $\mathcal{O}(1)$. Logarithmic and linear algorithms are good. Log-linear algorithms are acceptable: sorting, for example, is log-linear.[10] Polynomial algorithms, $\mathcal{O}(n^k)$ with $k > 2$, tend to be useful only for small problem sizes.

Exponential and factorial algorithms are useless except for very small problems. A problem whose best known solution has exponential or factorial complexity is called ***intractable***. Such problems must be solved by looking for approximations rather than exact results. One of the best-known intractable problems is TSP: the "travelling salesperson problem". A salesperson must make a certain number of visits and the problem is to find an ordering of the visits that minimizes some quantity, such as cost or distance. The only known way to find an exact solution is to try all possible routes and note the minimal route. If n visits are required, the number of possible paths is $\mathcal{O}(n!)$, making the exact solution infeasible if n is in the hundreds or even thousands.

Note that TSP is typical of problem descriptions. We are not really interested at all in travelling salespersons and, in any case, their actual problems (which might involve 20 visits at most) are easily solved. But TSP represents the generic problem of finding a minimal path in a weighted graph, and many practical problems can be put into this abstract form. One example is: find the quickest path for a drilling machine that has to drill several thousand holes in a printed-circuit board. 59

Figure 16 on the facing page shows the progress that has been made in solving three-dimensional elliptic partial differential equations. Equations of this kind must be solved for VLSI simulation, oil prediction, reactor simulation, airfoil simulation, and other significant problems. The difference between $\mathcal{O}(N^7)$ and $\mathcal{O}(N^3)$ corresponds to a factor of N^4, or a million times for a problem for which $N = 100$.

[10] Provided that you don't use bubblesort.

Function	Condition	Description
$\mathcal{O}(1)$		constant
$\mathcal{O}(\log n)$		logarithmic
$\mathcal{O}(\sqrt{n})$		square-root
$\mathcal{O}(n)$		linear
$\mathcal{O}(n \log n)$		log-linear
$\mathcal{O}(n^2)$		quadratic
$\mathcal{O}(n^3)$		cubic
$\mathcal{O}(n^k)$	$k > 1$	polynomial
$\mathcal{O}(a^n)$	$a > 1$	exponential
$\mathcal{O}(n!)$		factorial

Figure 15: Part of the complexity hierarchy

It is often claimed that hardware has improved more rapidly than software. For some problems, however, the improvements in software have been just as dramatic as those for hardware. Putting the two together, a modern supercomputer can solve differential equations more than a trillion ($10^6 \times 10^6 = 10^{12}$) times faster than was possible in 1945.

Method	Year	Complexity
Gaussian elimination	1945	$\mathcal{O}(N^7)$
SOR iteration (suboptimal parameters)	1954	$\mathcal{O}(N^5)$
SOR iteration (optimal parameters)	1960	$\mathcal{O}(N^4 \log N)$
Cyclic reduction	1970	$\mathcal{O}(N^3 \log N)$
Multigrid	1978	$\mathcal{O}(N^3)$

Figure 16: Solving three-dimensional elliptic partial differential equations (adapted from ***Numerical Methods, Software, and Analysis***, by John Rice (McGraw-Hill, 1983))

2.9 Example: Maximum Subsequence

The following problem arises in pattern recognition: find the maximum contiguous subvector of a one-dimensional vector (see (Bentley 1986, pp. 69–80)). The problem is trivial if all of the values in the vector are positive, because the maximum subvector is just the whole vector. If some of the values are negative, the problem is interesting. For example, given the vector 60

$$31 \quad -41 \quad 59 \quad 26 \quad -53 \quad 58 \quad 97 \quad -93 \quad -23 \quad 84$$

our algorithm should find the subvector

$$59 \quad 26 \quad -53 \quad 58 \quad 97$$

We assume that an empty subvector has sum 0. This implies that the maximum subvector can never be negative because, if we had a negative subvector, we could always replace it with the (larger) adjacent empty subvector.

There is an obvious solution: we can simply sum ***all possible*** subvectors and note which one has the largest sum. We assume that the given vector has N elements. We need three nested loops: one to choose the first element of the subvector, one to choose the last element, and one
61 to sum the elements in between. Figure 17 shows a solution based on this idea. It uses `max`, a standard C++ library function that returns the greater of its two arguments.

```
int maxSoFar = 0;
for (int i = 0; i != N; ++i)
{
    for (int k = i; k != N; ++k)
    {
        int sum = 0;
        for (int j = i; j <= k; ++j)
            sum += v[j];
        maxSoFar = max(maxSoFar, sum);
    }
}
```

Figure 17: Maximum subvector: first attempt

The algorithm of Figure 17 has complexity $\mathcal{O}(N^3)$. It is not efficient and, by inspecting it carefully, we can see how to do better. During each cycle of the outer loop, we can use a single loop to sum all subvectors starting at that point. This gives the second version of the algorithm,
62 with two nested loops and complexity $\mathcal{O}(N^2)$, shown in Figure 18.

```
int maxSoFar = 0;
for (int i = 0; i != N; ++i)
{
    int sum = 0;
    for (int j = i; j != N; ++j)
    {
        sum += v[j];
        maxSoFar = max(maxSoFar, sum);
    }
}
```

Figure 18: Maximum subvector: second attempt

Many, perhaps most, programmers would give up at this point and simply assume that quadratic complexity is the best that can be achieved. But, being more persistent, we will seek a better solution using invariants.

Here is a useful, general technique for solving problems with one-dimensional vectors: process the vector one element at a time, maintaining and updating as much information as is needed to proceed to the next step. Specifically, suppose we are about to process element `i`. What useful information can we obtain?

The first point to notice is that if we have a subvector "ending here", we can update it simply by adding `v[i]` to it. This will give us a new subvector "ending here" that we can keep if it is bigger than anything we have already seen.

The second point to notice is that we can remember the value of the largest subvector seen "so far" (this corresponds to what "we have already seen" in the previous sentence). The invariant that we need is:

`maxEndingHere`	$\equiv$	the largest subvector that ends here
`maxSoFar`	$\equiv$	the largest subvector we have seen so far

To make the invariant true initially, we set both variables to zero. When we examine `v[i]`, we note that `maxEndingHere` will not get smaller if `v[i]` > 0. Figure 19 shows the final version of the algorithm. This version requires time proportional to the length of the sequence. This is much better than the first version, which required time proportional to the ***cube*** of the length of the sequence. 63

```
int maxSoFar = 0;
int maxEndingHere = 0;
for (int i = 0; i != N; ++i)
{
    maxEndingHere = max(maxEndingHere + v[i], 0);
    maxSoFar = max(maxSoFar, maxEndingHere);
}
```

Figure 19: Maximum subvector: an efficient solution

References

Bentley, J. (1986). *Programming Pearls*. Addison-Wesley.

Dijkstra, E. W. (1976). *A Discipline of Programming*. Prentice-Hall.

Koenig, A. and B. E. Moo (2000). *Accelerated C++: Practical Programming by Example*. Addison-Wesley.