Mathematics 1262 (introduction to C++), Hilary 2013

Colm Ó Dúnlaing

March 31, 2014

Contents

1 Hello world

As always, we begin with ‘Hello world.’

C++ programs get data from keyboard/disc/etcetera, and write data to terminal/disc/etcetera. Writing data is called **output.**

In C++ the preferred style for output is through **cout**. For example

```
        /*
         * This C++ program prints a greeting
         */

#include <iostream>
        // Always needed.

using namespace std;
        // Almost always needed.

int main()
{
  cout << "hello" << endl << "It’s a nice day" << endl;

  return 0;
        // A convention.
}
```

Weird? The idea (see Conor Houghton’s notes) is that **cout** is like a pipe connecting the program to the terminal, and the << operator shifts data along the pipe. First hello, then a newline, then a greeting, and another newline.

Such a program should be stored in a file called `hello.cpp` or something similar. Next it must be **compiled:** a special program to convert it into ‘machine code’ which the computer can execute. The compiler on Unix machines is called

```
              g++
```

The default name for the machine code (for historical reasons) is **a.out**.

You can run it by just typing `a.out`.

```
%g++ hello.cpp
%a.out            (the next two lines are printed by the program)
hello
It’s a nice day
```

Advanced topic. If you want to give the machine code a more meaningful name, such as `myprog`,

```
g++ -o myprog hello.cpp
```

More advanced. What's the **# include<iostream>** needed for?

Because the input/output system isn't actually part of the C++ language, but a system that is stored in a 'library' and brought in as needed. **iostream** refers to a file which contains all the information `g++` needs to make sense of statements such as

```
  cout << "hello" << endl;
```

You can inspect the file if you want — it's extraordinarily complicated. (On Unix machines, `/usr/include/c++/...`)

Very advanced. What's the **using namespace std;** for?

Names are used all over the place, and the namespace is — used in very large projects — like a surname. Most of the time everything has the surname **std**. When you have `using namespace std;` the 'full name' of `cout` would be **std::cout**. You might have reasons to use `cout` as another name, and you could set things up so your own `cout` is `murphy::cout`.

The upshot is: if you leave out the `using namespace std;` you cannot use `cout, endl` on their own, but must use `std::cout, std::endl`.

Finally, that **return 0;** *statement.* The value is returned somehow to the 'operating system.' I don't know how to get that value. `return 0;` conventionally means 'this program ran with no problems' and nonzero means 'there were errors running this program.'

2 Variables

Like most programming languages, C++ programs obtain data from keyboard/disc/etcetera, store it in **variables**, operate on the variables, and write data to terminal/disc.

Really, a variable is a name for a piece of data. There are different kinds of data

- The minimal data is a *bit,* which can be 0 or 1. A group of 8 bits forms a *byte,* which is effectively the smallest piece of data — hence kilobyte, megabyte, gigabyte, terabyte.
- Character data — letters, digits, punctuation, etcetera, are stored under the ASCII encoding.
- An *integer* is stored in 4 bytes under a scheme which allows a range of about ± 2 billion.
- A *boolean* is an integer restricted to 0 or 1. Equivalent to a bit, it is probably stored as a byte.
- A *double-precision* floating-point number is stored in 8 bytes under various conventions which allow a range of roughly $\pm 2^{1000}$ **but** with 52 bits of precision.

  Very roughly, this allows numbers to be represented in the form

  $$\pm 10^e \times a.bc\ldots$$

  where $-300 < e < 300$ (very roughly) with about 15 or 16 decimal places of accuracy.

- There are also short and long integers, single-precision floating-point, and various ‘unsigned’ alternatives, which we won’t look at.

```
/*
 * Basic declarations involve ‘types’
 * bool, char, short, int, long, float, double,
 * and various ‘unsigned’ combinations.
 */

             // Example declarations
bool a;
int bc_2, Def, def;

            // These are variables. Names are case-sensitive.

            // statements are terminated by semicolon.

            // Also, constants; a name for pi...

const double pi = 3.14159;

             // Also, ‘enumeration types.’

enum Colour = { Red, Green, Blue };

// This gives a handy way of naming colours.
// Actually, the compiler converts them to integers.
```

3 Assignment statements and arithmetic

The most basic operation (beyond printing greetings) is **arithmetic assignnment**. Here are some declarations and assignment statements.

```
int a, b=0, c; // a,c have unpredictable values, b is intialised to 0

double d, e, f;

c = c+1;         // adds 1 to c, a new unpredictable value;
b = b+2;         // now b is 2
c = 3/4;         // c becomes ZERO: integer division rounds toward zero
d = 3/4;         // d becomes 0.00
e = 3.0/4;       // e becomes 0.75
```

Arithmetic statements use the operators `+,-,*,/,%` — the last is the remainder after division, or 'mod.'

Short-cuts include `a += b;` which adds `b` to `a`, and `c++, c--, ++c, --c` which increment and decrement `c` (the difference between `c++` and `++c` is a fine one.)

Remainder %. Integer division rounds towards zero.

```
100 / 13  evaluates to 7 and 100 % 13 evaluates to 9.

-100 / 13 evaluates to -7 and -100 % 13 evaluates to -9.
```

This is different from the mathematical convention where the remainder is always non-negative. Under the mathematical conventions, $-100 \div 13 = -8$ and $-100 \pmod{13} = 4$.

4 Compound statements and for-loops

Statements can be grouped together within braces $\{\}$, to form compound statements.

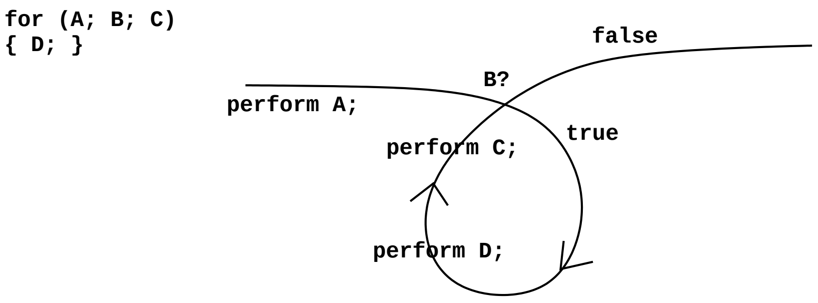

Figure 1: for loop

For-loops are the preferred method of repeating statements in C++. The above figure allows all the possibilities, but one shouldn't be too fancy. The statement A in the picture would be something like 'i=0,' initialising a variable i; the statement B would be something like 'i $<$ 10'; and the statement C should adjust the value of `i` in each iteration of the loop.

```
#include <iostream>

using namespace std;

int main ()
{
  int i;
  for ( i=0; i<10; ++i )
  {
    cout << "7 x " << i << " = " << 7 * i << endl;
  }
  return 0;
}
output.....
7 x 0 = 0
7 x 1 = 7
7 x 2 = 14
7 x 3 = 21
7 x 4 = 28
7 x 5 = 35
7 x 6 = 42
7 x 7 = 49
7 x 8 = 56
7 x 9 = 63
```

Arithmetic relations are

```
<    less than
<=   less than or equal
==   equal
        emphatically NOT =
        which means ASSIGNMENT
>    greater than
>=   greater than or equal
!=   not equal
```

Relations can be combined using

```
&&   logical and
         emphatically NOT & which
         means something else
||   logical or
         emphatically NOT | which
         means something else
!    logical negation
```

5 If-statements

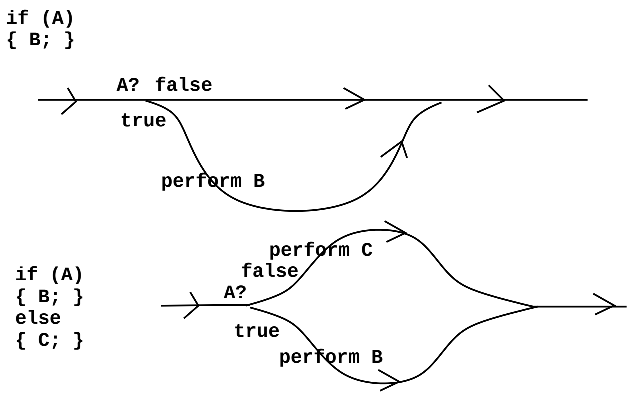

Figure 2: if and if-else statements

```
#include <iostream>

using namespace std;

int main ()
{
  int i;
  for ( i=0; i<10; ++i )
  {
   if ( i % 2 == 1 )
   {
    cout << "7 x " << i << " = " << 7 * i << endl;
   }
  }
  return 0;
}
output
7 x 1 = 7
7 x 3 = 21
7 x 5 = 35
7 x 7 = 49
7 x 9 = 63
```

6 While-loops

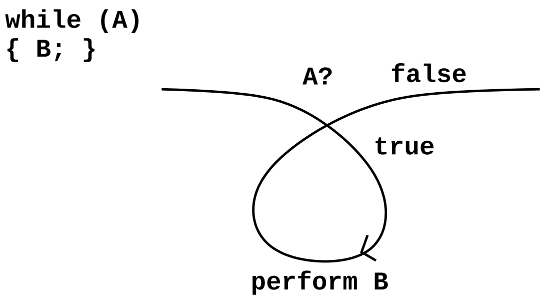

Figure 3: while loop

While-loops are obviously similar to for-loops. Their first important use is in controlling input. The following program reads numbers from the terminal (or more correctly, the keyboard) and echoes them. The expression **! cin.eof()** means: 'while the input stream **cin** has not reached end-of-file.'

When entering data from the terminal, end-of-data is signalled by

```
ctrl-D
```

at the beginning of a line (I don't think it works elsewhere). **Not** just a newline.

```
#include <iostream>

using namespace std;
```

```
main()
{

  int n;

  while ( ! cin.eof() )
  {
    cin >> n;
    cout << "The number you typed in was "  << n << endl;
  }

  return 0;
}
```

However, it doesn't work properly. Here is a sample run

```
% a.out
0
That was 0
1
That was 1
^D
That was 1
%
```

What's happening is that the end-of-file condition becomes true *only* after an ‘unsuccessful’ read. The following program does it properly.

```
#include <iostream>

using namespace std;

int main()
{

  int n;
  bool finished;

  finished = false;
  while ( ! finished )
  {
    cin >> n;
    if ( cin.eof ())
      finished = true;
    else
```

```
      cout << "That was " << n << endl;
  }

  return 0;
}
```

Sample run

```
%a.out
27182
That was 27182
314159
That was 314159
^D
%
```

7 Arrays, input, initialisation

Array subscripting in C++ uses square brackets. To declare an array of 100 elements, double-precision,

```
  double a[100]
```

- Array indexing begins at 0. C++ (and C) is unusual here. The highest index is 99.
- Each array entry occupies 8 bytes, so the total size of the array is 800 bytes.
- The array entries are stored in consecutive addresses

$$a_0, a_0 + 8, a_0 + 16, \ldots, a_0 + 99 * 8.$$

- Array indexing is *completely unchecked.* A reference to `a[-1]` would not be rejected, just converted to an address $a_0 - 8$. It may cause the program to crash at runtime. Again, a reference to `a[100]` converts to an address $a_0 + 800$, which is outside the range of the array.

In the example program below, note the test for end-of-data:

cin.eof()

```
#include <iostream>
// program reads numbers into an array and prints their sum.

using namespace std;

main()
```

```
{

  double a[100], next;
  int i, count;
  double total;

  count = 0;
  while ( count < 100 && ! cin.eof() )
  {
    cin >> next;
                              // corrected C++ code
    if ( ! cin.eof () )
    {
      a[count] = next;
      ++ count;
    }
  }

// At this point count gives the number of
// items stored in the array.  Note the precaution
// that count cannot be > 100.  Excessive input
// entries are ignored.

  total = 0;
  for ( i=0; i < count; ++i )
  {
    total += a[i];
  }

  cout << count << " items, total " << total << endl;

  return 0;
}
Compile and run (with a largeish file temp.  Notice how it's used)
%g++ array.cpp
%a.out < temp
100 items, total 9957.69
%
```

Various features of arrays can be extracted using for-loops. For example, instead of calculating the sum of the elements, one can calculate their maximum. This requires an if-statement.

```
// This is an incomplete piece of code.  It is assumed that the
// array a has been read in as above.  A double-precision variable
// maximum is assumed.

  if ( count == 0 )
    cout << "Nothing read in, maximum undefined" << endl;
  else
  {
    maximum = a[0];
    for ( i = 1; i<count; ++i )
    {
      if ( a[i] > maximum )
      {
        maximum = a[i];
      }
    }
    cout << count << " items read, maximum " << maximum << endl;
  }
...... compile and run as modified
% g++ maximum.cpp
% a.out < temp
100 items read, maximum 101
%
```

An obvious application of arrays is to store data for statistical analysis. Linear algebra is another obvious application. A third and less obvious use is for *tables.* For example,

```
  const int offset[12] = {0,3,3,6,1,4,6,2,5,0,3,5};
```

Notice that an array can be *initialised.* This is very useful. That particular array is useful in deducing the day-of-week from a given date (it is a running sum, modulo 7, of the lengths of months in an ordinary year.) There is an example program `weekday.cpp` which converts any date in this century to day-of-week.

8 Character strings

- A character in C is denoted 'A', 'b', '9', etcetera.
- There are some special characters;

  ```
  '\n' newline character (can be used in place of endl but not vice-versa)
  '\t' tab character
  '\0' null character.  This is not the full list.
  ```

- A character string is an *array* of characters.

- Since the size of an array is ignored, there must be another way to mark the end of a character string.
- The *null character* marks the end of a character string. Hence a string of length n is stored in $n+1$ bytes.
- Initialisation is possible in two styles

```
  char hello[100] = "hello";
  char goodbye[100] = {'g','o','o','d','b','y','e','\0'};
```

- For technical reasons,

```
  char * my_string;
```

  is another way of declaring character strings. However, unlike an array, no storage is reserved except by an **initialiser**.

- For example,

```
  const char * dayname[] = {"Sunday","Monday",... etcetera ... , "Saturday"};
```

  Note in this example that the size of the array is not specified. The compiler deduces it — i.e., size 7 — from the initialiser.

9 Addresses of array elements

Because C++ arrays have first index 0, the formula for array indexing is as follows.

```
Address of a[i] =
        address of a[0] +
                i * (element size in bytes)

In an array of size n, the total memory used by the array
is n * (element size in bytes)
```

Given an array declaration

```
  <type> <array name> [ <array size> ]  (= optional initialiser) ;
E.g.,
  double a [ 14 ];
```

The array size, i.e., the number of elements, is not necessary if an initialiser is given. The type can be `char, int, long, double, bool` etcetera.

- There is a ‘pseudo-function’ `sizeof()` which gives the size associated with certain types. For example, `sizeof(char)` is 1, `sizeof (int)` is 4, `sizeof (long)` (long integer) is 4 or 8 depending on the machine, `sizeof ( double )` is 8, sizeof (bool) appears to be 1.
- If the size of array items is s and the size of the array itself is n, then the total storage occupied by the array is sn. For the given example, the total array size is 112 bytes.
- For example, if the example array `a` begins at address 1234, then `a[10]` is stored at address $1234 + 10 * 8 = 1314$.

10 Functions and routines

For the past two weeks we have looked at C programs where all the code is in the part headed `main()`.

Large-scale C++ programs can run to tens of thousands of lines, possibly hundreds. They can’t all be stuffed between two braces following `main()`.

In fact, a program is usually separated into many ‘basic’ or ‘primitive’ procedures: deciding what is ‘primitive’ is the main part of the design process.

As usual we look at some silly examples.

Reconsider a program of the ‘hello world’ kind as one which prints a message: the primitive operation is to write a message. Accordingly the program includes a *routine* `message( bool x )`.

C++ allows *routines* which do things and *functions* which calculate things.

```
  void message ( bool x ) ...
        void means ROUTINE.  This does something.
  double average ( int n, double x[] )
        Here, double means double-precision-floating-point-valued FUNCTION.
        This computes something.

        Functions ‘return’ the value they compute using
        return statements.
```

```
#include <iostream>

using namespace std;

void message ( bool x )
{
  if ( x )
    cout << "Hello" << endl;
  else
    cout << "Goodbye" << endl;
}

main()
{
  message ( true );
  message ( false );

  return 0;
}
%a.out
Hello
Goodbye
%
```

The `average` example is much more interesting.

```
#include <iostream>

using namespace std;

double average ( int n, double x[] )
{
  int i;
  double total;

  total = 0;
  for ( i=0; i < n; ++i )
  {
    total += x[i];
  }

  return total / n;
}

main()
{

  double a[100], next;
  double av;
  int i, count;

  count = 0;
  while ( count < 100 && ! cin.eof() )
  {
    cin >> next;
    if ( ! cin.eof () )
    {
      a[count] = next;
      ++ count;
    }
  }

  av = average ( count, a );
  cout << "Average of " << count << " numbers is " << av << endl;

  return 0;
}


running:
%a.out < data/big
Average of 100 numbers is 1e+06
```

This looks pretty grotty. It's time to fine-tune our `cout << ...` statements. Unfortunately, I have the most primitive idea of how to do this.

The following changes produce a better output.

```
Add
#include <iomanip>
        // i/o manipulation !

and change the output statement to

  cout << setprecision(20) <<
        "Average of " << count << " numbers is " << av << endl;

This produces the output
Average of 100 numbers is 1000000.546875
```

11 Simulating routines and functions

Here to 'simulate' a routine means (given its arguments) to write down the sequence of values taken by its variables, and thereby compute what it computes. For example

```
int xxx ( int n )
{
  int i, x;
  x = 0;
  for ( i=0; i<n; ++i )
  {
    x += i;
  }
  return x;
}
--------
xxx (4):

    n    i     x
    4
               0
         0
               0
         1
               1
         2
               3
         3
               6
returns 6
```

It is important to tabulate the values in the order they are created, i.e., not to have them side-by-side

```
    n    i     x
    4    0     0
         1     1
         2     3
         3     6
returns 6
```

It is too confusing. **Exercise:** what does this compute, given $n \geq 0$?

Another example (every routine is named `xxx`)

```
int xxx ( int m, int n )
{
  int i, x;
  x = 0;
  for ( i=0; i<m; ++i )
  {
    x += n;
  }
  return x;
}
```

Exercise: simulate `xxx ( 3, 4 )`. What does `xxx ( m, n)` compute in general, given $m \geq 0$?

And another

```
int xxx ( int m, int n )
{
  int x;
  x = 0;
  while ( m > 0 )
  {
    if ( m % 2 == 1 )
      x += n;
    m = m/2;
    n = n * 2;
  }

  return x;
}
```

This is called *Russian peasant multiplication.*

Notice that the arguments m and n are ‘used’ as local variables. This is safe because they are copies of expressions in the calling program. If they were call-by-reference then it would be a mistake to use them this way.

The idea behind Russian peasant multiplication can be useful in practice. For example, a modified form of this routine can be used to calculate m^n — still not useful — and to calculate A^m if A is a square matrix. This is useful as the number of multiplications is proportional to $\log_2 m$ rather than m.

```
int xxx ( int m, int n )
{
  int x,y,z;
  x = m;
  y = n;
  while ( y > 0 )
  {
    z = x % y;
    x = y;
    y = z;
  }
  return x;
}
```

This is Euclid’s gcd algorithm, an old favourite.

12 2-dimensional arrays

Two-dimensional arrays are a logical extension of ordinary arrays. For example,

```
  int b[3][4];
```

has 12 entries and size 48 (in bytes). In general, if the 'sizeof' array entry type is s, m 'rows' and n 'columns,' the array occupies smn bytes.

C++ convention dictates that `b` is equated to an array of arrays, that is, an array of 3 arrays of 4 ints. To calculate positions in the array, we need the fact that each 'row' of the array is 4 ints, so it has size 16. If `b` starts at address e, then `b[i][j]` has address $e + 16i + 4j$.

Strangely, each 'row' `b[i]` has a **value,** and that value is the address where the row begins. This is consistent with the i-th row being a 1-dimensional array.

In general, given 'sizeof' array entries is s, starting address is e, and there are n 'columns,' the address of the `[i][j]` entry is

$$e + ins + js$$

For example, suppose

```
  long int c[5][9];
```

and `c` begins at address 4000.

Sizeof long int: 4 or 8 bytes depending on machine.
Sizeof row: 36 or 72 bytes.

Suppose long ints occupy 8 bytes.
Value of `c[3]` is $4000 + 3 \times 72 = 4216$.
Address of `c[2][5]` is $4000 + 2 \times 72 + 5 \times 8 = 4184$.

13 Call by reference; also, overloading

In this section we touch on a subtle and very important question about function/routine arguments: 'Call by value' versus 'call by reference.' The upshot is that

- By default, arguments are call-by-value.
- Array arguments are effectively call-by-reference.
- C++ (unlike C) allows call-by-reference.

Suppose our program needs to read in a matrix using a routine such as

```
void read_matrix ( int m, int n, double a[10][10])
```

- An array of fixed size is passed in which to store the matrix.

- The arguments m and n are supplied as the 'correct' dimensions. obviously they can be no more than 10 in either dimension. There is no simple way to pass 2-dimensional arrays of variable dimension.

- Routine arguments are usually 'call by value.' All arguments `m, n, a` are call by value. This means that the routine `read_matrix()` works with *copies* of whatever the calling program passed. If `read_matrix()` changed them, it is only copies which would be changed, and the changes would be forgotten when the routine returned.

- Shouldn't the same hold for the matrix `a`? No. An array variable in C++ is stored as an *address*, that is, the address of the first array element. This is very economical: no matter how big an array is passed to a subroutine, all that is actually passed is the 4 or 8 bytes giving its starting address.

  So an assignment to `a[i][j]` within `read_matrix()` becomes an assignment to '`row i, column j` in an array whose starting address is ...' This is an array in the calling program.

Unlike C, C++ has an explicit *call-by-reference* style for passing arguments:

```
void read_matrix ( int & m, int & n, double a[10][10] )
```

Now a copy of m is not passed: a copy of the *address* of m is passed. Any operations on m within `read_matrix()` affect the *calling variable.*

In particular, `read_matrix()` can read m and n and communicate them to the calling program.

Function overloading. C++, unlike C, allows the same function name to be used several times, so long as the argument lists are distinguishable. For example (not that this does anything, but you see the idea)

```
#include <iostream>

using namespace std;

double sum ( int n, double x[] )
{
}

int sum ( int n, int x[] )
{
}

int main()
{
  return 0;
}
```

14 Variables all over the place

- Variables declared within routines/functions, including call-by-value arguments, are *automatic*, only existing within the run of the routine. They are stored on the *runtime stack* in a *stack frame* for the routine.
- Variables can be declared at the top of the `.cpp` file. These are *global*, and are visible from all routines.
- In C++, variables can be declared almost anywhere.
- Variables can also be declared within *blocks*: a block is a group of statements { `...`} between braces. They are local to the block.
- Variables can be declared at the *head* of for-loops, such as

```
for ( int i = 0; i<100; ++i )
{
  ....
}
```

Example.

```
#include <iostream>

using namespace std;

int n = 0;

void a ()
{
  cout <<  "a " << n << endl;
}

void b ( int n )
{
  cout <<  "b " << n << endl;
}

void c ( int k )
{
  n = k;
}

int main ()
{
  int i, n, j;
  n = 25;

  a();
  b(33);
  c(55);
  a();

  for ( i=0; i<n; ++i )
  { }

  cout << "i " << i << endl;

  for ( j = 4; j < 10; ++j )
  {
    int i = j*j;
    cout << "i " << i << " ";
  }

  cout << endl;


  cout << "i " << i << endl;
  cout << "j " << j << endl;

  j = 100;
  cout << "j " << j << endl;

  for ( int j = 4; j < 10; ++j )
  {
    int i = j*j;
    cout << "i " << i << " ";
  }

  cout << endl;

  cout << "i " << i << endl;
  cout << "j " << j << endl;
}
%a.out
a 0
b 33
a 55
i 25
i 16 i 25 i 36 i 49 i 64 i 81
```

```
i 25
j 10
j 100
i 16 i 25 i 36 i 49 i 64 i 81
i 25
j 100
```

15 Random numbers

A *random number generator* is a system for producing a long sequence of *pseudo-random* numbers. These are not random, because first the sequence is always the same, and second the sequence is generated by some fairly simple rule. In C++

```
#include <cstdlib>
rand() produces a pseudo-random number between 0
and RAND_MAX, a predefined constant
and
srand() ‘sets a seed.’
```

One can create uniformly-distributed double-precision numbers by taking an integer random number and dividing it by `RAND_MAX + 1`.

This needs to be done carefully,

else `RAND_MAX + 1` will be negative!

The sequence is *intended* to be the same each time. To get a different sequence each time, one can use a ‘random seed.’ Look at `rand_double.cpp`:

```
#include <iostream>
#include <cstdlib>
#include <sys/time.h>

using namespace std;

static bool seeded = false;

        /*
         * cstdlib is needed for rand()
         * sys/time.h (the .h means a C file) is used for
         * setting a random ‘seed’ by the microseconds part
         * of the system clock.
         * Probability against repetition roughly a million to 1.
         */

static void seed ()
```

```
{
  struct timeval tv;
  gettimeofday ( & tv, NULL );
  srand ( tv.tv_usec );
}
```

Explanation. `gettimeofday()` is a C library function. Structures will be discussed very soon. The **&** in `gettimeofday( & tv, NULL` means that an *address* is passed — C does not have call-by-reference.

Here is the `timeval` structure (from `man gettimeofday` in Unix)

```
    struct timeval {
             long    tv_sec;         /* seconds since Jan. 1, 1970 */
             long    tv_usec;        /* and microseconds */
     };
```

So what `seed()` does is to 'read the clock' and use the microseconds as a seed for the random number generator.

By the way, that **static** means 'private.'

```
double rand_double()
{
  double divisor;

  divisor = RAND_MAX;
  divisor += 1;
        // This is to ensure double-precision calculations.
        // RAND_MAX+1 is negative in integer arithmetic.

  if (! seeded )
  {
    seed ();
    seeded = true;
  }

  return rand() / divisor;
}
```

15.1 Coins and dice.

As an exercise, we can use random number generators to simulate the throwing of two dice. Here is `dice.cpp`

```
#include <cstdlib>
#include <sys/time.h>

using namespace std;

static bool seeded = false;

static void seed ()
{
  struct timeval tv;
  gettimeofday ( & tv, NULL );
  srand ( tv.tv_usec );
}

int rand_6()
{
  if (! seeded )
  {
    seed ();
    seeded = true;
  }

  return (rand() % 6) + 1;
}

main( int argc, char * argv[])
{
  int n = atoi ( argv[1] );
  int i,j,k;
  double count[13];
  double table[13] =
  { 0, 0, 1.0/36, 2.0/36, 3.0/36, 4.0/36, 5.0/36,
        6.0/36, 5.0/36, 4.0/36, 3.0/36, 2.0/36,
        1.0/36};

  for (i=0; i<13; ++i)
    count[i] = 0;

  for (i=0; i<n; ++i)
  {
    j = rand_6();
    k = rand_6();
    count[ j+k ] += 1;
  }

  cout << "out of " << n << " throws\n";
  for ( i=2; i<=12; ++i)
  {
    cout << i << " came up a proportion of " << count[i]/n <<
        " times (probability " << table[i] << ")\n";
  }

}
```

This worked well on one machine

```
On a PC running Linux, the result of ‘dice’ was:
out of 10000 throws
2 came up a proportion of 0.0273 times (probability 0.0277778)
3 came up a proportion of 0.0558 times (probability 0.0555556)
4 came up a proportion of 0.0876 times (probability 0.0833333)
5 came up a proportion of 0.1108 times (probability 0.111111)
6 came up a proportion of 0.1426 times (probability 0.138889)
7 came up a proportion of 0.1681 times (probability 0.166667)
8 came up a proportion of 0.1335 times (probability 0.138889)
9 came up a proportion of 0.1081 times (probability 0.111111)
10 came up a proportion of 0.0864 times (probability 0.0833333)
11 came up a proportion of 0.0525 times (probability 0.0555556)
12 came up a proportion of 0.0273 times (probability 0.0277778)
```

But look at what happened on the maths machines:

```
On boole (and turing) the result was
out of 10000 throws
2 came up a proportion of 0 times (probability 0.0277778)
3 came up a proportion of 0.1081 times (probability 0.0555556)
4 came up a proportion of 0 times (probability 0.0833333)
5 came up a proportion of 0.219 times (probability 0.111111)
6 came up a proportion of 0 times (probability 0.138889)
7 came up a proportion of 0.3341 times (probability 0.166667)
8 came up a proportion of 0 times (probability 0.138889)
9 came up a proportion of 0.225 times (probability 0.111111)
10 came up a proportion of 0 times (probability 0.0833333)
11 came up a proportion of 0.1138 times (probability 0.0555556)
12 came up a proportion of 0 times (probability 0.0277778)
```

What’s the difference? As a clue, we do a coin-tossing simulation (`evens.cpp`)

```
#include <iostream>
#include <cstdlib>

using namespace std;

int main()
{
  int i;

  for ( i=0; i<10; ++i )
  {
    if ( i>0 )
    { cout << " "; }

    cout << rand() % 2;
  }
  cout << endl;

  return 0;
}
```

Again, compare answers from a PC and from boole:

```
pc:
1 0 1 1 1 1 0 0 1 1
boole:
0 1 0 1 0 1 0 1 0 1
```

So the `boole` output is not at all random. This means that a *linear congruential* generator is being used to produce $r_1, r_2, r_3, \ldots$:

$$r_{n+1} = ar_n + b \pmod{R}$$

where a and b are constants and R is `RAND_MAX + 1`. The point is that R is a power of 2, and if x_n is $r_n \pmod{2}$ then

$$x_{n+1} = a'x_n + b' \pmod{2}$$

where a', b' are the residues of a and b, modulo 2. Since

$$a'x + b' \pmod{2} = \begin{cases} 0 \\ 1 \\ x \\ x+1 \end{cases}$$

there is no room for randomness. This caused the problem also in `dice.cpp`. In consequence:

Careful about using remainders. Linear congruential generators are *not* random in the low bits. Evidently other kinds of generator are used on Linux PCs.

How can one compensate for this problem? Basically, to produce random numbers in the range $0 \ldots n-1$, get a random *real-valued* number s in the interval $[0,1)$, and return sn.

```
#include <iostream>
#include <cstdlib>
        // coin.cpp
        // Produces 10 random bits.  Not degraded
        // by linear congruential generators.

using namespace std;

int randbit()
{
  const double scale = 1.0 / (1.0 + RAND_MAX );
        // This is a reliable way to scale
        // It forces RAND_MAX to be converted
        // to double before adding 1.0
        // Note: RAND_MAX + 1 (integer arithmetic)
        // is -2^31

  double drand;

  drand = rand();
        // random number, converted to integer
  drand *= scale;
        // scaled to the interval [0,1)
  return ( drand >= 0.5 );
        // uniform [0,1) has probability 1/2 of
        // being >= 1/2
}

int main()
{
  int i;

  for ( i=0; i<10; ++i )
  {
    if ( i>0 )
    { cout << " "; }

    cout << randbit();
  }
  cout << endl;
  return 0;
}
```

16 Command-line arguments

When you run a program, `a.out`, say, anything between `a.out` and end-of-line is a *command-line argument*.

For example,

```
a.out  infile outfile
a.out 10000
```

- The command-line arguments are character strings
- They do not contain blanks (unless quotes are used)
- They are available to the program
- They are character strings, but can be converted to integer or double using the built-in functions `atoi(), atof()`

They are available to the program as follows.

- Instead of `int main()`, write `int main ( int argc, char * argv[] )`
- `argc` gives the number of command-line arguments, *including the program name* such as `a.out`.
- `argv[0]`, a character string, is the name of the program
- For $1 \leq i < argc$, `argv[i]` is the i-th command-line argument.

```
a.out in_file out_file
argv[0]: a.out
argv[1]: in_file
argv[2]: out_file

a.out 10000
argv[0]: a.out
argv[1]: 10000
        This is a character string.
        To use it, write something like

        n = atoi ( argv[1] );
```

```
or maybe

        x = atof ( argv[1] );

To use atof() or atoi(), you need
#include <cstdlib>
```

17 Structures

In programming it is important to collect data in usable units. This can be done in C++ (rarely: C++ has much better ways of doing it) using **structures**. For example, without structures we might write functions of 3-vectors like

```
double norm ( double x, double y, double z )...
```

(if one chooses not to use arrays). Now a structure with 3 components can be established as a new *type*

```
typedef struct { double x,y,z; } VEC3;

main()
{
  VEC3 a;
  a.x = 1; a.y = 2; a.z = 3;
        // this is how to get the components
        // of VEC3
  cout << a.x << " " << a.y << " " << a.z << endl;
}
```

17.1 Complicated details: please skip

Unfortunately, we must go further with this.

```
  VEC3 *a;
```

declares a variable `a` to be the *address* of a `VEC3`. It has *pointer type.* (Compare with `char * argv[]`).

This will not be covered in lectures, but the notes are retained (see below).

17.2 For example, matrices.

Structures are used to 'package' data. We have seen some examples involving matrices. But routines for matrices generally need three items:

- The height, call it m
- The width, call it n
- A 2-dimensional array such at `double a[10][10]`

It would make good sense to define a structure such as

```
typedef struct
{
  int height, width;
  double entry[10][10];
} MATRIX;
```

Now, at least, the height and width and matrix proper are kept in the same place, and you could refer to them as follows

```
  MATRIX mat;
  ...
  ... mat.height ...
  ... mat.width ...
  ... mat.entry[i][j]
```

This is restrictive, since height and width cannot exceed 10, and if less than 10 there is wasted space. Structures are really a C feature, not important in C++, and in C one can overcome these restrictions using tricks with pointers.

17.3 Continuing complicated details: please skip

They are accessed as shown below. (The combined character $->$ is supposed to resemble an arrow which 'points' to where an item is stored.

```
  a->x = 1; a->y = 2; a->z = 3;
```

In connection with pointer types, C++ provides a way to *create* a memory region which can hold a `VEC3`: **new**.

```
  VEC3 * a = new VEC3;
```

In practice, variables of structure type are rare; pointer types are more common.

```
#include <iostream>
#include <cmath> // for sqrt

using namespace std;

typedef struct {double x,y,z;} VEC3;

void print_vec ( VEC3 * a )
{
        // Prints without newlines
  cout << a->x << " " << a->y << " " << a->z;
}

VEC3 * make_vec ( double x, double y, double z )
        // ‘makes’ a vector with the given components.
{
  VEC3 * vec = new VEC3;
        // space has been reserved
  vec->x = x; vec->y = y; vec->z = z;
        // values are copied
  return vec;
}

void delete_vec ( VEC3 * vec )
{
        // to avoid ‘memory leaks.’ A technicality.
  delete vec;
}
```

Cross product, dot product, determinant:

```
VEC3 * cross_prod ( VEC3 * a, VEC3 * b )
{
  return
  make_vec (
    a->y * b->z - a->z * b->y,
    a->z * b->x - a->x * b->z,
    a->x * b->y - a->y * b->x );
}

double dot_prod ( VEC3 * a, VEC3 * b )
{
  return a->x * b->x + a->y * b->y + a->z * b->z ;
}


double det ( VEC3 * a, VEC3 * b, VEC3 * c )
{
  double d;
  VEC3 * bc = cross_prod ( b, c );
  d = dot_prod ( a, bc );
  delete_vec ( bc );
  return d;
}
```

```
int main()
{
  VEC3 * a, * b, * c, * d;
  double len;
  a = make_vec ( 1, 2, 3 );
  b = make_vec ( 4, 5, 6 );
  c = make_vec ( 7, 8, 8 );
  d = cross_prod ( a, b );


  cout << "a "; print_vec ( a ); cout << endl;
  len = sqrt ( dot_prod ( a, a );
  cout << "length " << len << endl;
  cout << "b "; print_vec ( b ); cout << endl;
  cout << "c "; print_vec ( c ); cout << endl;
  cout << "a x b "; print_vec ( d ); cout << endl;

  cout << "(a x b) dot c " << dot_prod ( d, c ) << endl;
  cout << "det(a,b,c) " << det ( a,b,c ) << endl;

  return 0;
}
```

18 Classes

Object-oriented programming is where the data is organised into groups of *objects*. C++ is intended for this style of programming. It adds greatly to the power of a language.

What separates C++ from C is the notion of **class**, an idea which probably appeared first in the language SIMULA67. An object is an *instance* of a class.

C++ regards the C `struct, union,`[1] C++ classes as closely related, but ‘structs’ and ‘unions’ only contain data. Classes contain data and also functions which operate with and on that data.

C is a ‘small’ language and in the space of a term it is possible to learn almost all of it. C++ may also be small, but classes are very sophisticated and take a lot of getting used to. As usual, we take a silly example to start.

```
#include <iostream>

using namespace std;

typedef class Boa
{
  public:
    void speak ();
    void toggle ();
    void showbit();
  private:
    int bit;
} Boa;
```

Definition. An incomplete function or routine declaration such as `speak();` which is followed by a semicolon, not {...code...}, is called a **function or routine prototype**. It describes the argument and return types, which are often needed before the code has been supplied.

What do public and private mean? Try

```
int main()
{
  Boa a;
  cout << a.bit << endl;
  return 0;
}
```

Note. *Members of classes are identified using the dot notation just as with* `struct`*s.* A member can be a variable, as with `struct` (a ‘field’ in C), or a function or routine.

This won’t compile.

```
%g++ boa_0.cpp
boa_0.cpp: In function ’int main()’:
boa_0.cpp:12: error: ’int Boa::bit’ is private
boa_0.cpp:20: error: within this context
```

[1] `Unions` were used in C to save space. They are probably obsolete. In C++ they are unnecessary because of ‘class inheritance.’ (I think.)

So, make `bit` public.

```
typedef class Boa
{
  public:
    void speak ();
    void toggle ();
    void showbit();

    int bit;
} Boa;
```

This time it did compile, and the output was

```
-1218875404
```

So obviously class objects aren't initialised. How about

```
typedef class Boa
{
  public:
    void speak ();
    void toggle ();
    void showbit();

    int bit = 0;
} Boa;
```

This doesn't compile

```
boa_1.cpp:12: error: ISO C++ forbids initialization of member 'bit'
boa_1.cpp:12: error: making 'bit' static
boa_1.cpp:12: error: ISO C++ forbids in-class initialization of
non-const static member 'bit'
```

So this doesn't work.[2] We'll come back to the initialisation problem later, and get on with the other pieces. We have to include code for `speak(), toggle(), showbit()`. These can be written almost like ordinary functions. The only difference is a prefix **Boa::**.

```
void Boa::speak ()
{
```

[2]I tried to get a rationale for this: googling revealed that C++ doesn't allow such initialisation but Java and C# do. One concludes that its implementation was considered a nuisance by the C++ designers, but would not have been impossible.

```
  if ( bit )
    cout << "Hello\n";
  else
    cout << "Goodbye\n";
}
```

Strictly speaking, `bit` should be `bool`, but C++ and C both accept general integer values as truth values. Any nonzero integer is interpreted as true: only 0 is false.

```
void Boa::toggle ()
        // toggle changes 0 to 1 and 1 to 0
{
  bit = 1 - bit;
}

void Boa::showbit()
{
  cout << bit << endl;
}

int main()
{
  Boa a;

  a.speak();
  a.showbit();
  a.toggle();
  a.showbit();
  a.speak();

  return 0;
}
```

Why the `showbit()`? Because `bit` is private, so it can't be seen from outside, whatever that means. The object itself has full access to the member `bit` which it *owns.* This program compiles, and works, except for the initialisation problem.

```
Hello
-1218101260
1218101261
Hello
```

Initialisation is handled through *constructors*. We need to add one more routine. It is declared in a slightly different way from normal routines, and it is named `Boa`.

Here is a full program.

```
#include <iostream>

using namespace std;

typedef class Boa
{
  public:
    Boa(); // constructor
    void speak ();
    void toggle ();
    void showbit();
  private:
    int bit;
} Boa;

void Boa::speak ()
{
  if ( bit )
    cout << "Hello\n";
  else
    cout << "Goodbye\n";
}

void Boa::toggle ()
{
  bit = 1 - bit;
}

void Boa::showbit()
{
  cout << bit << endl;
}

Boa::Boa()
{
  bit = 1;
}

int main()
{
  Boa a;
  int i;
  cout << "a.showbit () initially " ; a.showbit();
  for (i=0; i<5; ++i)
  {
    a.speak();
    a.toggle();
  }
  cout << "a.showbit () finally " ; a.showbit();

  return 0;
}

%a.out
a.showbit () initially 1
Hello
Goodbye
Hello
Goodbye
Hello
a.showbit () finally 0
```

---

19 Casts and mixed arithmetic expressions

19.1 Casts

- When assigning a value to a variable, the types should match, with some exceptions in the case of numerical values.
- One can assign The type of a variable is clear: it has to be declared. For the type of an expression, it is not clear. However, the types of constants are generally easy to recognise.

`'\n'`	`character`
`"hello"`	`character string`
`-45`	`int`
`1.23`	`double`
`true`	`bool`

- A double value can be assigned to an int, and vice-versa. Conversion of int to double is direct, that is, 3 becomes 3.0. Double to ints are **rounded towards zero,** so 1.23 becomes 1 and -1.23 becomes -1.
- An expression can be converted to another type using **casts**. The syntax is

```
   (  ... type .... )   expression
such as
   ( double ) 1;
```

- Also, expressions can have subexpressions of different types. For example,

  ```
   1 + 2.34
  ```

  is 3.34. More about this later.

We now have a clean solution to that difficult RAND_MAX problem: this constant is the maximum positive integer value, and if we add 1, we get the minimum negative integer value. Cast it to `double`, and add 1.

```
#include <iostream>
#include <cstdlib>

using namespace std;

int main()
{

  const int bad = 1 + RAND_MAX;
        // The compiler issues a warning, but goes ahead.

  const double rm_p_1 = 1 + (double) RAND_MAX;

  cout << "bad " << bad << endl;
  cout << "rand_max + 1 " << rm_p_1 << endl;

  return 0;
}
------------------
%g++ temp.cpp
temp.cpp: In function int main():
temp.cpp:9: warning: integer overflow in expression
alcom5% a.out
bad -2147483648
rand_max + 1 2.14748e+09
%
```

19.2 Expressions of mixed arithmetic type

An expression can be a combination of other expressions using $+, -, *, /, \%$. Where two subexpressions of different types are combined,

- Surprisingly, `char`s are promoted to ints.

  This can be problematic if there is ‘sign extension.’ A char with a face value > 127 has ‘high order bit 1’ and on some machines (including the maths machines) it is converted to a negative integer.

- bools are converted to ints.

- ints are converted to doubles.

- Floats are *always* converted to doubles in any arithmetic calculation.

Going back to

```
const double rm_p_1 = 1 + (double) RAND_MAX;
```

the expression on the right mixes int with double; the 1 is converted to double before adding.

One last example.

```
1 - 2.3 - 4 is evaluated from left to right
1 - 2.3 is -1.3
-1.3 -  is -5.3

1 + 2/3  becomes 1 + 0 then 1
1 + 2.0/3 = becomes 1 + 0.666667 then 1.666667
```

20 Class Vec3

This class is for the usual vector computations in $\mathbb{R}^3$. It’s a variation of `struct VEC3`, turned into a class.

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

using namespace std;


typedef class Vec3
{
  public:
    Vec3();
    Vec3( double, double, double );
    double x();
    double y();
    double z();
    void copy ( Vec3 );
    Vec3 cross_prod ( Vec3 );
    double dot_prod ( Vec3 );
    double norm();
    double det ( Vec3, Vec3 );
    void print();
  private:
    double x1, x2, x3;
} Vec3;
```

- There are *two* constructors, distinguishable by their argument lists. This is an example of *overloading.*
- `Vec3 (double, double, double)` is enough of a prototype. You don't need to write `Vec3 (double x, double y, double z)`.
- This may be a bad idea, but `x()` is supposed to be the x-component, privately `x1`. This is a way of publishing the components.
- `copy()` is supposed to copy the components of another vector into the current one.
- `cross_prod ()` returns a `Vec3`. It's used in the following style:

  ```
  c = a.cross_prod ( b )
  ```

- Likewise `dot_prod()`. Determinant is

  ```
  a.det ( b, c)
  ```

- And `norm()` takes no arguments, just returning the norm of the current vector:

  ```
  n = a.norm();
  ```

- Finally,

  ```
  #include <cmath>
  ```

  is needed because a mathematical function `sqrt()` is used.

20.1 Code for the class member functions and routines

```
Vec3::Vec3()  // argument-free constructor
{
  x1 = x2 = x3 = 0;
}
```

```
Vec3::Vec3 ( double xx, double yy, double zz )
                // Constructs vector with given components
{
  x1 = xx; x2 = yy; x3 = zz;
}

double Vec3::x()
                // the x-component
{
  return x1;
}

double Vec3::y()
                // the y-component
{
  return x2;
}

double Vec3::z()
                // the z-component
{
  return x3;
}

void Vec3::copy ( Vec3 other )
                // Copies other to here; a routine;
                // overwrites current values:
                // doesn't return a new Vec3.
{
  x1 = other.x(); x2 = other.y(); x3 = other.z();
}


                // For comparison, code from the ‘C-style’ struct
                // VEC3 is shown, ‘commented out.’

//VEC3 * cross_prod ( VEC3 * a, VEC3 * b )
//{
//  return
//  make_vec (
//    a->y * b->z - a->z * b->y,
//    a->z * b->x - a->x * b->z,
//    a->x * b->y - a->y * b->x );
//}
```

```
Vec3 Vec3::cross_prod ( Vec3 other )
                // returns a.cross_prod ( b ) ....
{
  double xx, yy, zz;

  xx = y() * other.z() - z() * other.y();
  yy = z() * other.x() - x() * other.z();
  zz = x() * other.y() - y() * other.x();

  return Vec3(xx, yy, zz);
}

//double dot_prod ( VEC3 * a, VEC3 * b )
//{
//  return a->x * b->x + a->y * b->y + a->z * b->z ;
//}

double Vec3::dot_prod ( Vec3 other )
{
  return
    x1 * other.x() + x2 * other.y() + x3 * other.z();
}

double Vec3::norm ()
{
  return sqrt ( x1*x1 + x2*x2 + x3*x3 );
}

//double det ( VEC3 * a, VEC3 * b, VEC3 * c )
//{
//  double d;
//  VEC3 * bc = cross_prod ( b, c );
//  d = dot_prod ( a, bc );
//  delete_vec ( bc );
//  return d;
//}

double Vec3::det ( Vec3 b, Vec3 c )
{
  return
    dot_prod ( b.cross_prod ( c ) );
}

//void print_vec ( VEC3 * a )
//{
```

```
//        // Prints without newlines
//  cout << a->x << " " << a->y << " " << a->z;
//}

void Vec3::print ()
{
  cout << x1 << " " << x2 << " " << x3;
        // no newline
}

int main()
{
  Vec3 a(1,2,3), b(4,5,6), c(7,8,8);
  Vec3 d = a.cross_prod ( b );

  cout << "a "; a.print(); cout << endl;
  cout << "b "; b.print(); cout << endl;
  cout << "c "; c.print(); cout << endl;
  cout << "a x b\n"; d.print(); cout << endl;
  cout << "norm " << d.norm() << endl;
  cout << "(a x b) dot c\n" << d.dot_prod ( c ) << endl;
  cout << "det(a,b,c)\n" << a.det(b,c)<< endl;

  return 0;
}
-------------------output
% a.out
a 1 2 3
b 4 5 6
c 7 8 8
a x b
-3 6 -3
norm 7.34847
(a x b) dot c
3
det(a,b,c)
3
%
```

21 Files

Up to now, input has been taken from the keyboard (or indirectly, using $<$, from a file, and output has been to the screen or indirectly, using $>$, to a file.

This is the tip of the iceberg. C++ offers very elaborate file-handling facilities.

We shall mention only a few. For further information, consult the cplusplus.com reference web page.

- **ifstream** is a type (actually a class) which allows input from a named file.
- **ofstream** for output.
- A file must be *opened* — connected to a named file.
- Output files must be *closed*, otherwise their contents will be lost.
- Operations used with **cin, cout** can be used here. In particular, <<**,** >>**,** and member function **eof()**.
- An alternative way of reading data is using the member function **getline (buffer, count)**. This transfers input into the array 'buffer' as a string. It reads up to the next newline, if any, but with a limit on the number of characters read.

  Buffer is a character array — 200 characters, say; and count gives a maximum number of characters allowed. Under the string conventions, a string ends with '
  0' — so at most 199 characters are read.
- You need to include <`fstream`>.

Here is an example. It uses a **string** class, of which we shall see more later.

Why? It was mentioned before that the >> operator can be used to read data (from **cin**) into character arrays. This is dangerous, because there is no control on the number of characters read. The C++ **string** class allows for strings of any length. Reading this way into a C++ string is perfectly safe.

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

using namespace std;

int main( int argc, char * argv[])
{
  ifstream input;
  char buffer[200];
  bool finished;
  string s;

  input.open ( argv[1] );

  finished = false;
  cout << "-----------------INPUT line by line------------------" << endl;

  while ( ! finished )
  {
    input.getline ( buffer, 200 );
    if ( input.eof () )
      finished = true;
    else
      cout << buffer << endl;
  }

  input.close();

  input.open ( argv[1] );
  finished = false;

  cout << endl << "-----------------INPUT word by word------------------"
        << endl;

  while ( ! finished )
  {
    input >> s;
    if ( input.eof () )
      finished = true;
    else
      cout << s << endl;
  }

  return 0;
}
```

Sample output:

```
-----------------INPUT line by line------------------
Ladle Rat Rotten Hut

Wants pawn term, dare worsted ladle gull hoe lift wetter
murder inner ladle cordage, honor itch offer lodge, dock,

-----------------INPUT word by word------------------
Ladle
Rat
Rotten
Hut
```

```
Wants
pawn
term,
dare
worsted
ladle
gull
hoe
lift
wetter
murder
inner
ladle
cordage,
honor
itch
offer
lodge,
dock,
```

22 Templates, constants, etcetera

22.1 Templates

C++ allows generalised functions and classes. Class template definitions are as follows

```
template <class T> class <class name> { ... };
```

where <T> is a 'type parameter.' We shall not be concerned with writing generic classes or functions. But we shall use them. Using a class template is simple:

```
#include <vector>

  vector <int> x;
```

uses a *class template* `vector`, and declares `x` to be a *vector of int*s.

Less easy to understand is a function or routine template. Here is one we shall use.

```
template <class RandomAccessIterator>
void sort ( RandomAccessIterator first, RandomAccessIterator last );
```

This is a template because `RandomAccessIterator` is an incomplete class — that is, not all its functions are coded. Sorting works through a kind of magic. More later.

22.2 Constants in classes

One can use constants within class definitions.[3] For example, suppose one created a class `String` for character strings. This would be ridiculous, because there is already a class `string` which can do almost anything with character strings.

```
#include <iostream>

using namespace std;

typedef class String
{
  public:
    String();
    static const int max = 1000;
    String ( char * buf );
    void print();
  private:
    char store[ max ];
} String;

String::String ()
{
  store[0] = '\0';
}

int main()
{
  String s;
  cout << "s.max is " << s.max << endl;
  cout << "String::max is " << String::max << endl;

  return 0;
}
```

It is necessary to add the keyword ‘static’ to `max`. Otherwise the compiler says that ISO C++ forbids initialisation.

Notice something rather odd:

It is possible to use `String::max` *independent of any variable of class* `String`.

For example, in the ‘real’ string class, there is an enormous constant `npos`, apparently the same as `RAND_MAX`, which is the maximum possible length of strings. It is used for default values. It can be referred to via

[3]Many features can be declared inside classes, such as `typedef`.

```
...
#include <string>
...
... string::npos ...
```

23 Object-oriented programming

Object-oriented programming is where the data is organised into *objects*, larger units than the basic `int, char, double,` etcetera. The class mechanism in C++ is intended for object-oriented programming.

C++ comes with a *standard template library*, a collection of prefabricated classes. These include **string, vector, set, map,** and others. Used properly, a great deal of programming effort can be avoided. The web-page

`http://www.cplusplus.com/reference`

is extremely useful, our 'bible.' It has a full description of the various classes discussed below, and plenty more besides.

- Reading a text file line-by-line (**getline()**) or word by word (>>).
- Reading words and storing them in a `vector`.

  Note. The `vector` class is a *class template*. One can have vectors of ints, of doubles, of strings, and so on: this must be specified, as for example in

  ```
  #include <vector>  // can't use vector objects without this
  ...
    vector<string> v;
  ...
  ```

- Reading words, reducing them (`#include <cctype>`) by removing non-alphanumeric characters, and storing.
- Reading words, storing, and sorting the vector: then the words appear in sorted order.
- Reading words, storing them in a **set**, and printing them using an iterator. They are printed in sorted order, without repetitions.

  `set` is a class template: one uses `set <string>` and so on.

23.1 Pairs

There is a *class template* **pair** which enables one to construct ordered pairs. For example,

```
  pair <int, int> p;
```

defines a variable p. You can get its components through

```
  p.first
  p.second
```

Example.

```
#include <iostream>
#include <vector>

using namespace std;


int main()
{

  vector < pair <int,int> > v;
  int i;
  for (i=0; i<10; ++i)
   v.push_back ( pair <int, int> ( i, i*i ) );

  cout << "OUTPUT-----------\n";
  for (i=0; i<10; ++i)
    cout << v[i].first << ' ' << v[i].second << endl;

  return 0;
}
OUTPUT -------------------------
0 0
1 1
2 4
3 9
4 16
5 25
6 36
7 49
8 64
9 81
```

23.2 Replacing words

We shall write a program to replace occurrences of one word by another. This shows the power of the **string** class.

We use the following member functions.

- `string(buffer)`: constructor; constructs a string containing the characters in `buffer` (up to '\0').
- `find ( word )` returns the starting position of the earliest occurrence of the word within the string.
- If the word does not occur in the string, `find` returns `string::npos`, an effectively 'infinite value.' (Probably $2^{32} - 1$).
- `replace ( pos, len, newword)` removes `len` characters beginning at `pos` in the string, inserting `newword` in its place.

```
#include <iostream>
#include <cstdlib>

using namespace std;

int main( int argc, char * argv[] )
{
  bool finished;
  char buffer[ 200 ];

  string oldword, newword, str;

  oldword = string ( argv[1] );
  newword = string ( argv[2] );

  finished = false;
  while ( ! finished )
  {
    cin.getline ( buffer, 200 );
    if ( cin.eof() )
      finished = true;
    else
    {
      str = string ( buffer ); // convert
      int i = str.find ( oldword );
      while ( i < string::npos )
      {
        str.replace ( i, oldword.length(), newword );
        i = str.find ( oldword );
      }

      cout << str << endl;
    }
  }

  return 0;
}
-----------------------------------
a.out groin-murder GREENMUTTER < ../data/ladle
Ladle Rat Rotten Hut

Wants pawn term, dare worsted ladle gull hoe lift wetter
murder inner ladle cordage, honor itch offer lodge, dock,
florist. Disk ladle gull orphan worry putty ladle rat cluck
wetter ladle rat hut, an fur disk raisin pimple colder
Ladle Rat Rotten Hut.

Wan moaning, Ladle Rat Rotten Hut's murder colder inset.
"Ladle Rat Rotten Hut, heresy ladle basking winsome burden
barter an shirker cockles. Tick disk ladle basking tutor
cordage offer groinmurder hoe lifts honor udder site offer
florist. Shaker lake! Dun stopper laundry wrote! Dun
stopper peck floors! Dun daily-doily inner florist, an
yonder nor sorghum-stenches, dun stopper torque wet
strainers!"

"Hoe-cake, murder," resplendent Ladle Rat Rotten Hut, an
tickle ladle basking an stuttered oft.

Honor wrote tutor cordage offer GREENMUTTER, Ladle Rat
Rotten Hut mitten anomalous woof.

" Wail, wail, wail! " set disk wicket woof, "Evanescent
Ladle Rat Rotten Hut! Wares are putty ladle gull goring
wizard ladle basking?"

"Armor goring tumor GREENMUTTER's," reprisal ladle gull.
"Grammar's seeking bet. Armor ticking arson burden barter
an shirker cockles."

"O hoe! Heifer gnats woke," setter wicket woof, butter
taught tomb shelf, "Oil tickle shirt court tutor cordage
offer GREENMUTTER. Oil ketchup wetter letter, an den
```

```
O bore!"

Soda wicket woof tucker shirt court, an whinney retched a
cordage offer GREENMUTTER, picked inner windrow, an sore
debtor pore oil worming worse lion inner bet. En inner
flesh, disk abdominal woof lipped honor bet, paunched honor
pore oil worming, an garbled erupt. Den disk ratchet
ammonol pot honor GREENMUTTER's nut cup an gnat-gun, any
curdled ope inner bet.

Inner ladle wile, Ladle Rat Rotten Hut a raft attar
cordage, an ranker dough ball. "Comb ink, sweat hard,"
setter wicket woof, disgracing is verse. Ladle Rat Rotten
Hut entity betrum an stud buyer GREENMUTTER's bet.

"O Grammar!" crater ladle gull historically, "Water bag
icer gut! A nervous sausage bag ice!"

"Battered lucky chew whiff, sweat hard," setter
bloat-Thursday woof, wetter wicket small honors phase.

"O Grammar, water bag noise! A nervous sore suture
anomolous prognosis!"

"Battered small your whiff, doling," whiskered dole woof,
ants mouse worse waddling.

"O Grammar, water bag mouser gut! A nervous sore suture
bag mouse!"

Daze worry on-forger-nut ladle gull's lest warts. Oil offer
sodden, caking offer carvers an sprinkling otter bet, disk
hoard hoarded woof lipped own pore Ladle Rat Rotten Hut an
garbled erupt.

Mural: Yonder nor sorghum stenches shut ladle gulls stopper
torque wet strainers.
```

24 Iterators

Definition *Paraphrased* from the C++ reference:
An *iterator* is any object that,

- pointing to an object within a range (collection) of objects

- has the ability to **iterate** through the elements of that collection
- using a set of operators (including at least $++$ (increment) and dereference ($*$) operators).

Dereference: In C, if a is the address of a piece of data, then $*a$ is the piece of data. 'Dereferencing a' means getting the piece of data stored at a.

Iterators provide uniform ways of traversing objects in various different orders. Classes such as `string, vector, set, map,` include them. They are also used to define the range for generic sorting routines.

They generally involve highly complex expressions such as

```
pair <map<string,int>::iterator,  bool> ret;
```

This defines a variable `ret` whose type is a *pair* (see above) of items, the first being a

```
map<string,int>::iterator
```

Explanation.

`map` is a class template. Within any class one can have public and private `typedef`s. For example,

```
#include <iostream>

using namespace std;

typedef class A
{
  public:
    typedef int I;
} A;

int main()
{
  A::I i = 234;
  cout << i << endl;
  return 0;
}
```

Anyway, it is ok to use

```
map<string,int>::iterator
```

as the name for a type.

The general style for iterators is

```
for ( ITERATOR TYPE it = v.begin(); it != v.end(); ++it )
```

For example

```
  for ( set<string>::iterator it = v.begin();
        it != v.end(); ++it )
```

Also, an iterator returns a *pointer* to an item each time. It was mentioned before, but only in passing: if i is of type ‘pointer to x’ then $*i$ is the item pointed to. This is important in the example below.

Forward and reverse iterators are provided. Here again is a small program to print out words taken from the input. It removes non-alphanumeric characters. It uses *reverse* iterators, so the results

```
#include <iostream>
#include <string>
#include <cctype>
#include <set>
#include <algorithm>

using namespace std;

int main()
{
  bool finished;
  string str;
  string trunc;
  set<string> v;

  finished = false;
  while ( ! finished )
  {
    cin >> str;
    if ( cin.eof() )
      finished = true;
    else
    {
      trunc = string();
      for ( int i=0; i<str.length(); ++i )
        if ( isalnum ( str[i] ) )
          trunc.push_back ( str[i] );
      if ( trunc.length() > 0 )
        v.insert ( trunc );
    }
  }

  for ( set<string>::reverse_iterator i = v.rbegin();
        i != v.rend(); ++ i )
  {
    cout << * i << endl;
  }

  return 0;
}
```

Running it on our 'Ladle Rat' sample text, we get

```
your
yonder
wrote
worsted
worse
worry
worming
woof
etcetera
```

Further points.

- An iterator is used to traverse collections.
- They are used also to pinpoint locations in sophisticated objects, 'containers,' such as sets or maps; cf `find()`.
- They contain plenty of information. In fact, they contain enough information to sort the object completely. For example,

  ```
  sort (v.begin(), v.end())
  ```

24.1 Sorting

To illustrate the last,

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main()
{ int list[6] = {3,1,4,1,5,9};
  vector < int > v;

  for (int i = 0; i<6; ++i)
```

```
  { v.push_back ( list[i] );
  }
  sort ( v.begin(), v.end() );
  for ( int i=0; i<v.size(); ++i )
  { cout << v[i] << ' ';
  }
  cout << endl;

  return 0;
}
```

24.2 Maps

There is a class template **map**<**domain_type, codomain_type**> which is very useful for storing features or properties of things.

- It is a little awkward to use, since it is essentially a **set** of **pair**<**domain_type, codomain_type**>, and when inserting, one creates and inserts a pair.
- Iterators are returned when consulting the map. Such an iterator can be treated as a *pointer* to a pair.

Note. Recall that if `x` is of type `pair <...>` then its two components are `x.first` and `x.second`.

If x is of type `pointer to pair...` then its two components are

```
(*x).first        and        (*x).second
```

There is an alternative notation:

```
x->first        and        x->second
```

```
#include <iostream>
#include <map>

using namespace std;

int main()
{
  map <int,int> mp;
  map <int,int> :: iterator it;

  for ( int i=0; i<10; ++i )
  { mp.insert ( pair<int,int> ( i, i*i ) ); }
  cout << "OUTPUT-----------\n";

  for ( int i=0; i<10; ++i )
  { it = mp.find ( i );
    cout <<  it->first << " mapsto " << it->second << endl;
  }

  return 0;
}
OUTPUT-----------
0 mapsto 0
1 mapsto 1
2 mapsto 4
3 mapsto 9
4 mapsto 16
5 mapsto 25
6 mapsto 36
7 mapsto 49
8 mapsto 64
9 mapsto 81
```

25 STL: useful features

	string	vector	set	map
begin(), end(), rbegin(), rend()	string	vector	set	map
size()	string =length()	vector	set	map
operator[]	string	vector		map
push_back()	string	vector		
find() default	string npos		set end()	map end()
substr() replace()	string			
insert			set	map

- `#include <string>`
  `or vector, set, map, or algorithm`

- Only string is a class; vector, set, and map, are class templates:

  ```
  vector <type>, set <type>
  map <type1, type2>
  ```

- All the classes contain the given iterators.

- All classes have functions `size()`, with an alternative `length()` for strings.
- Operator [] works with strings, vectors, and maps. With strings and vectors it works as expected.

  ```
  m [ x ]
  ```

  returns the value stored for `x`; if `x` is not there, a pair `(x, d)` is stored where `d` is hopefully a default value.
- push_back works on strings and vectors, adding to the end.
- find() works on strings, sets, and maps.
  - On a string `s`, `s.find ( str )` returns the leftmost position where an occurrence of string `str` begins, default **string::npos** (or `s.npos`).
  - On a set `s`, `s.find ( val )` returns an **iterator** from which the entry in `s` can be dereferenced: default `s.end()`.
  - On a map `m`, `m.find ( val )` returns an **iterator** to a **pair** `(val, y)` can be dereferenced; default `m.end()`.
- `substr()` and `replace()` work on strings.
  - `s.substr ( i, ell)` returns the substring of length `ell` beginning at location `i`.
  - `s.substr ( i, ell)` returns the substring of length `ell` beginning at location `i`.
  - `s.replace ( i, ell, newword )` replaces the substring of length `ell` beginning at location `i`, by `newword`.
- `insert` inserts an element into a set or an ordered pair into a map.

25.1 Sorting

There is also

```
#include <algorithm>
sort ( first, last )
eg
sort ( v.begin(), v.end() );
```

which takes *iterators* pointing to first and ‘beyond last’ elements to be sorted. Iterators contain a lot of information, and it is enough to sort the object without naming the object.

26 Recursion

Recursion in C++ and in C works because of the *runtime stack*. In this way different copies of local variables can be held together.

A *recursive* routine or function is one which 'calls itself.' (Or A calls B which calls A, etcetera). The most popular example is the factorial function.

```
#include <iostream>

using namespace std;

int fac ( int n )
{
  if ( n == 0 )
  {
    return 1;
  }
  else
  {
    return n * fac ( n-1 );
  }
}
int main()
{
  cout << "4! is " << fac ( 4 ) << endl;

  return 0;
}
OUTPUT:
4! is 24
```

Simulating this function. *Indentation* helps show how recursion works. The Asterisks are for alignment.

```
main calls
  fac (4):
  * n is 4. It calls
  *   fac (3):
  *   * n is 3. It calls
  *   *   fac (2)
  *   *   * n is 2. It calls
  *   *   *   fac (1)
  *   *   *   * n is 1. It calls
  *   *   *   *   fac (0)
  *   *   *   *     n is 0.
  *   *   *   *     returns 1 to...
  *   *   *   * here which returns
  *   *   *   n * 1 = 1 to
  *   *   here which returns
  *   *   n * 1 = 2 to
  *   here which returns
  *   n * 2 = 6 to
  here which returns
  n * 6 = 24 to
here which prints
4! is 24
```

It was mentioned some time back that *local variables and routine/function arguments are stored on the runtime stack.* This 'stacking' action is suggested by the indentation. You see that the same variable n has several copies sitting in different places on the stack.

Another example is to add together elements of an array.

```
#include <iostream>

using namespace std;

int sum ( int i, int j, int x[] )
{
  if ( i > j )
    return 0;
  else
    return
      sum ( i, j - 1, x ) + x[j];
}
int main()
{
  int a[6] = {3, 1, 4, 1, 5, 9 };
  cout << "sum of 6 numbers is " << sum (0, 5, a ) << endl;

  return 0;
}
OUTPUT
sum of 6 numbers is 23
```

Simulation:

```
main calls
  sum (0, 5, a)
  * i = 0, j = 5. Calls
  *   sum (0, 4, a)
  *   * i = 0, j = 4. Calls
  *   *   sum (0, 3, a)
  *   *   * i = 0, j = 3. Calls
  *   *   *   sum (0, 2, a)
  *   *   *   * i = 0, j = 2. Calls
  *   *   *   *   sum (0, 1, a)
  *   *   *   *   * i = 0, j = 1. Calls
  *   *   *   *   *   sum (0, 0, a)
  *   *   *   *   *   * i = 0, j = 0. Calls
  *   *   *   *   *   *   sum ( 0, -1, 0)
  *   *   *   *   *   *   * i = 0, j = -1. Returns 0 to
  *   *   *   *   *   here which returns 0 + x[0] to
  *   *   *   *   here which returns x[0]+x[1] to
  *   *   *   here which returns x[0]+x[1]+x[2] to
  *   *   here which returns x[0]+x[1]+x[2]+x[3] to
  *   here which returns x[0]+x[1]+x[2]+x[3]+x[4] to
  here which returns x[0]+x[1]+x[2]+x[3]+x[4]+x[5] to
here which prints the total.
```

A recursive version of ‘Russian peasant multiplication.’

```
#include <iostream>

using namespace std;

int xxx ( int m, int n )
{
  int temp;

  if ( m == 0 )
    return 0;
  else
  {
    temp = xxx ( m/2, n );
    if ( m % 2 == 0 )
      return temp + temp;
    else
      return temp + temp + n;
  }
}
```

```
int main()
{
  cout << "xxx ( 5, 70 ) is " << xxx ( 5, 70 ) << endl;

  return 0;
}
OUTPUT
xxx ( 5, 70 ) is 350
```

Simulation

```
main calls
  xxx ( 5, 70 )
  * m = 5, n = 70, temp=?. Calls
  *   xxx ( 2, 70)
  *   * m = 2, n = 70, temp = ?. Calls
  *   *   xxx ( 1, 70)
  *   *   * m = 1, n = 70, temp = ?.  Calls
  *   *   *   xxx ( 0, 70) which returns 0 to
  *   *   here which sets temp to 0 and returns
  *   *   * 0 + 0 + 70 to
  *   here which sets temp to 70 and returns
  *   * 70 + 70 + 0 to
  here which sets temp to 140 and returns
  * 140 + 140 + 70 = 350 to
here which prints
xxx ( 5, 70 ) is 350
```

27 IEEE standard

In the mid-1980s the IEEE defined a standard for floating-point calculations.

Nonzero single-precision floating-point numbers are interpreted as numbers of the form

$$\pm 1.b_1 b_2 \ldots b_{23} \times 2^e, \quad -126 \le e \le 127$$

The bitstring $1b_1 \ldots b_{23}$ is called the *significand* (or *mantissa*). An exponent of -127 is possible; with $b_1 \ldots b_{23}$ all zero, this represents $\pm$ *zero*.

Exponent 128 is also possible, with $b_1 \ldots b_{23}$ all zero, this represents $\pm\infty$. If not all zero, this is *not a number*, `NaN`.

Excluding ∞ and `NaN`, we shall call any number representable in this system, just *representable*. *Machine epsilon* ϵ_{mach} is 2^{-23}. The smallest representable number > 1 is

$$1 + 2^{-23} = 1 + \epsilon_{\text{mach}}.$$

The IEEE standard requires that, given representable numbers x, y and an operation $\circ$ (add, subtract, multiply, divide), the hardware computes $x \circ y$ exactly rounded. It also mentions square root and computing remainders.

In double-precision arithmetic, the exponent range is from -1022 to 1023, with -1023 and 1024 used for zero and infinity and `NaN`; there are 52 rather than 23 binary digits after the point, and

$$\epsilon_{\text{mach}} = 2^{-52}.$$

Pretty well all modern processors conform to the IEEE standard.

28 Accuracy of summation

The IEEE standard is concerned with *relative error.* That is, if X is exact and $\tilde{X}$ is the computed approximation,

$$\frac{X - \tilde{X}}{X}$$

or, put differently,

$$\tilde{X} = X(1 + \delta)$$

where δ is small. The significance of this appears in connection with computing the variance.

(28.1) Proposition *Suppose n and δ_i , $1 \le i \le n$, are given, where $n\epsilon_{mach} < 1$, and $|\delta_i| \le \epsilon_{mach}$ for $1 \le i \le n$. Let $\prod_i (1 + \delta_i)^{\pm 1} = 1 + \theta_n$. Then*

$$|\theta_n| \le \gamma_n. \quad \blacksquare$$

(28.2) Lemma *Unless n is absurdly large, the rounding error in evaluating*

$$x_1 + \ldots + x_n$$

is bounded by $\gamma_{n-1} \sum |x_i|$.

Proof. Evaluating gives

$$\begin{gathered}\sum x_i = \\ ((((x_1 + x_2)(1 + \delta_1) + x_3)(1 + \delta_2) + \ldots + x_{n-1})(1 + \delta_{n-2}) + x_n)(1 + \delta_{n-1}) = \\ x_1(1 + \theta_{n-1}) + x_2(1 + \theta_{n-1}) + x_3(1 + \theta_{n-2}) + \ldots + x_n(1 + \theta_1) = \\ (\sum x_i) + x_1\theta_{n-1} + x_2\theta_{n-1} + x_3\theta_{n-2} + \ldots + x_n(1 + \theta_1)\end{gathered}$$

Here θ_r is a product of r terms of the form $(1 + \delta_i)$ where $|\delta_i| \le \epsilon_{\text{mach}}$. By the above two lemmas, $|\theta_r| \le \gamma_{n-1}$ for $1 \le r \le n$, and the overall error is bounded in absolute value by

$$(|x_1| + \ldots + |x_n|)\gamma_{n-1}. \quad \blacksquare$$

Unfortunately, since the signs of the terms can vary, there is no bound connecting the error to $\sum x_i$: the best we can hope for is relate the error to $\sum |x_i|$.

This crops up when we use the following formula for variance

$$\frac{\sum_i x_i^2 - n\overline{x}^2}{n-1}$$

As shown with suitable data (where the average is very large), this can be inaccurate. Explanation: the *relative* error in computing $\sum_i x_i^2$ is still very small, but relative to the true variance it is much larger.

On the other hand, if $\sum_i (x_i - \overline{x})^2/(n-1)$ is calculated directly, the error is very small in comparison with the correct value. This is guaranteed by the IEEE standard.

29 Numerical accuracy

An important quantity in floating-point error analysis is

$$\gamma_n = \frac{n\epsilon_{\text{mach}}}{1 - n\epsilon_{\text{mach}}}.$$

For example, Gaussian elimination can be related to the so-called LU factorisation of an $n \times n$ matrix A

$$A = LU$$

where L is lower triangular and U is upper triangular. Computation of L and U is related to Gaussian elimination, and Gaussian elimination amounts to solving

$$\tilde{L}\tilde{U}X = Y$$

Where $\tilde{L}$ and $\tilde{U}$ are computed approximations to L and U. Moreover, there is an error bound:

$$|LU - \tilde{L}\tilde{U}| \leq \gamma_n |\tilde{L}||\tilde{U}|$$

where $|A|$ is the matrix of absolute values of A.

Pivoting is not involved. If pivoting is involved, the LU factorisation is applied not to A but to a row-permuted version of A. This is a very important difference.

For example, solve (with e very small, $\epsilon_{\text{mach}}/4$, say),

$$\begin{bmatrix} e & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

Gauss-Jordan elimination without pivoting yields

```
e  1  2         e     1      2       e    1    2
1  1  1         0  1-1/e   2-1/e     0  -1/e -1/e
```

The last matrix is with correct rounding; $2 - 1/e = -1/e$ correctly rounded, so $y = 2$; substituting, $x = 0$. With pivoting,

```
1  1  1          1    1         1       1   1   1
e  1  2          0    1-e       2-e     0   1   2
```

whence $y = 2$ and $x = -1$. In exact arithmetic,

$$y = \frac{2-e}{1-e} \quad x = 1 - \frac{2-e}{1-e}$$

and the computed result is accurate. The inaccuracy — in the absence of pivoting — can be related to the LU factorisation. With exact arithmetic,

$$LU = \begin{bmatrix} 1 & 0 \\ 1/e & 1 \end{bmatrix} \begin{bmatrix} e & 1 \\ 0 & 1 - 1/e \end{bmatrix} = \begin{bmatrix} e & 1 \\ 1 & 1 \end{bmatrix} = A$$

whereas

$$\tilde{L}\tilde{U} = \begin{bmatrix} 1 & 0 \\ 1/e & 1 \end{bmatrix} \begin{bmatrix} e & 1 \\ 0 & -1/e \end{bmatrix} = \begin{bmatrix} e & 1 \\ 1 & 0 \end{bmatrix} \quad \text{and}$$
$$|\tilde{L}||\tilde{U}| = \begin{bmatrix} 1 & 0 \\ 1/e & 1 \end{bmatrix} \begin{bmatrix} e & 1 \\ 0 & 1/e \end{bmatrix} = \begin{bmatrix} e & 1 \\ 1 & 2/e \end{bmatrix}.$$

So $\gamma_n|\tilde{L}||\tilde{U}|$ allows a large error in the $(2,2)$ position.

30 Linear algebra package

There are downloadable linear algebra packages. One is `armadillo`, and here is an example using it.

The example is to produce a certain kind of layout for a certain kind of graph. A graph is represented abstractly as a list of (unordered) pairs.

For example, six vertices and nine edges

```
1,2    1,4    1,5
2,3    2,6    3,4
3,6    4,5    5,6
```

Drawn one way, this graph has vertices 1,2,3,4 on the outer face.

Given a layout of the outer vertices (they must form a convex polygon), the layout can be extended *barycentrically* to the other vertices, meaning that every internal vertex is the average (centroid) of its neighbours.

This can be expressed with matrices. The above example will be summarised in the following input file

```
6 4
0 -1.5 -1
```

```
1 1.5  -1
2 1.5  1
3 -1.5 1
9
0 1 0 3 0 4
1 2 1 5 2 3
2 5 3 4 4 5
```

For the above example, the following matrix is formed[4]

```
   1.0000         0         0         0         0         0
        0    1.0000         0         0         0         0
        0         0    1.0000         0         0         0
        0         0         0    1.0000         0         0
  -1.0000         0         0   -1.0000    3.0000   -1.0000
        0   -1.0000   -1.0000         0   -1.0000    3.0000
```

and used to solve two sets of equations. The first characterises the x-values, the other the y-values. The right-hand sides are (transposed)

$$\begin{bmatrix} -1.5 & 1.5 & 1.5 & -1.5 & 0 & 0 \end{bmatrix}^T$$
$$\begin{bmatrix} -1 & -1 & 1 & 1 & 0 & 0 \end{bmatrix}^T$$

Here is the code.

```
#include <iostream>
#include <vector>
#include <armadillo>

        //Input format
        //n b  no.of vertices, no of boundary vertices
        //boundary point: index x-value y-value (b times)
        //m no. of edges
        //edge: vertex from vertex to
        //
        //Produces coordinates for all the points and
        //extra information suitable for drawing the
        //graph.
        //
        //A nuisance job is to rearrange the vertex
        //indices to put the boundary points first.
```

[4] No it isn't. That was an old version of the program.

```
using namespace std;

int main()
{
  int n, b, m, ix, iy, count;
  double x, y, scale;
  vector <pair<int,int> > edge;
  vector <int> perm1, perm2;
  arma::imat adjacent;
  arma::mat mainmat;
  arma::mat inverse;
  arma::colvec xvals, yvals;
  arma::colvec xsol, ysol;

  cin >> n >> b;

  perm1 = vector<int>(n,-1);
  perm2 = vector<int>(n,-1);

  xvals = arma::zeros<arma::colvec>(n,1);
  yvals = arma::zeros<arma::colvec>(n,1);

  for (int i=0; i<b; ++i )
  {
    int j;
    cin >> j >> x >> y;
    xvals(i) = x; yvals(i) = y;
    perm1[i] = j;
    perm2[j] = i;
    ++ count;
  }

  count = 0;
  for (int j = 0; j<n; ++j)
  {
    if ( perm2[j] < 0 )
    {
      perm2[j] = b + count;
      ++ count;
    }
  }        // this will allow for rearrangement

  cin >> m;

  adjacent = arma::zeros<arma::imat> (n,n);
```

```
mainmat = arma::eye<arma::mat> (n, n);

for ( int i =0; i<m; ++i )
{
  cin >> ix >> iy;
  adjacent (perm2[ix], perm2[iy]) =
  adjacent (perm2[iy], perm2[ix]) = 1;
      // see how the matrix is rearranged
}

for ( int i=b; i<n; ++ i )
{
  count = 0;
  for (int j=0; j<n; ++j )
  {
    if ( adjacent (i,j) )
    {
      ++ count;
      mainmat (i,j) = -1;
    }
  }
  mainmat (i,i) = count;
}

solve ( xsol, mainmat, xvals );
solve ( ysol, mainmat, yvals );

cout << n <<  ' ' << b << endl;

for (int i=0; i<n; ++i)
  cout << xsol(i) << ' ' << ysol(i) << endl;

cout << m << endl;
for ( int i=0; i < n; ++i )
{
  for ( int j=i+1; j<n; ++j )
  {
    if ( adjacent(i,j) )
    {
      cout << i << ' ' << j << ' ' << xsol(i) << ' ' << ysol(i) << ' '
      << xsol(j) << ' ' << ysol(j) << endl;
    }
  }
}
```

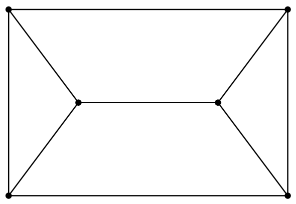

Figure 4: 6-vertex graph with barycentric embedding

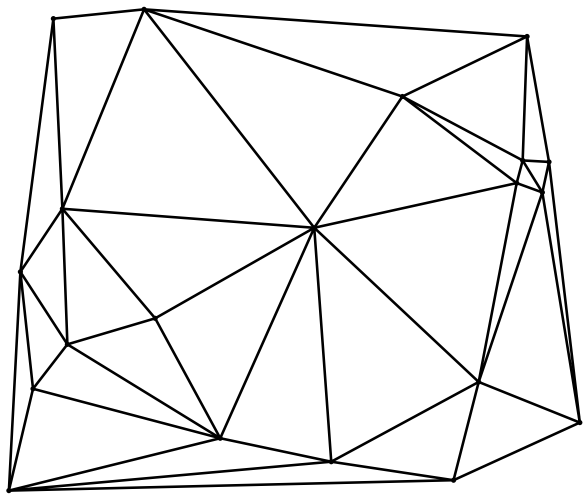

Figure 5: Delaunay triangulation of 20 points

```
  return 0;
}
```

This can be viewed in various ways. There are various graph-drawing packages such as ‘neato.’ Or if one knows postscript, one can generate a postscript file, as was done for the given example.

Figures 5 to 9 give bigger examples.

31 Various topics

31.1 Operator overloading

Just as one can have several functions of the same name, one can re-interpret operators such as $+, -, ==$, etcetera. For example.

```
#include <iostream>

using namespace std;

typedef class Vec2
{
```

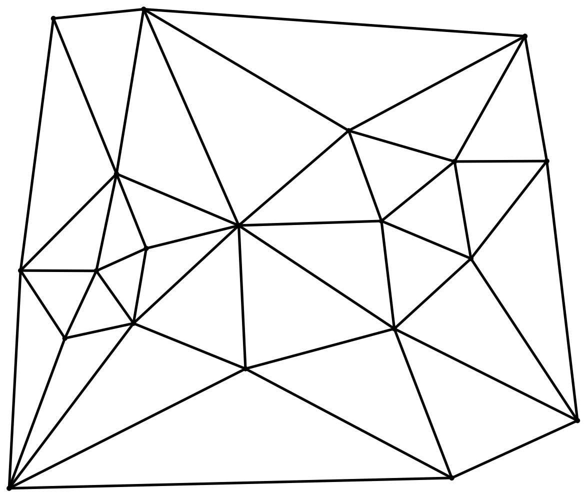

Figure 6: barycentric embedding of same graph

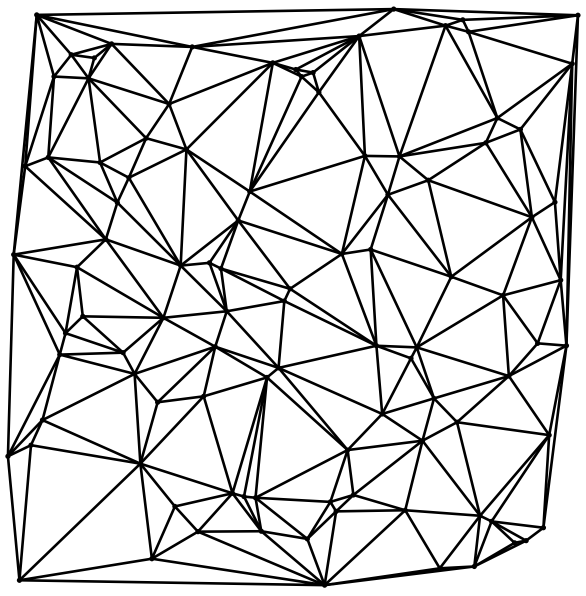

Figure 7: Delaunay triangulation of 100 points

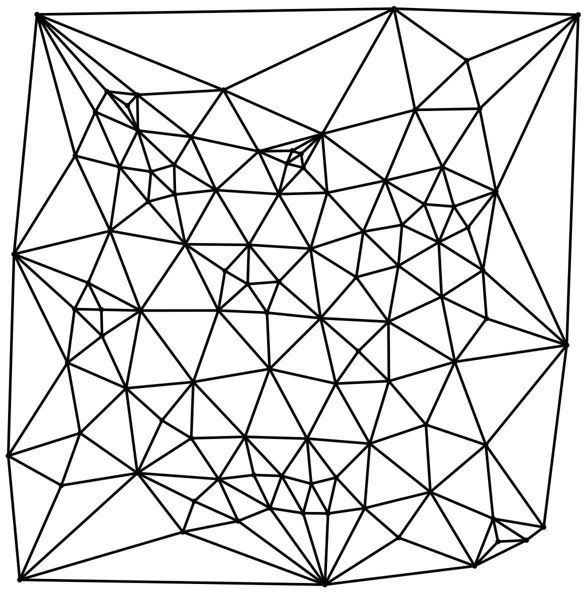

Figure 8: barycentric embedding of above graph

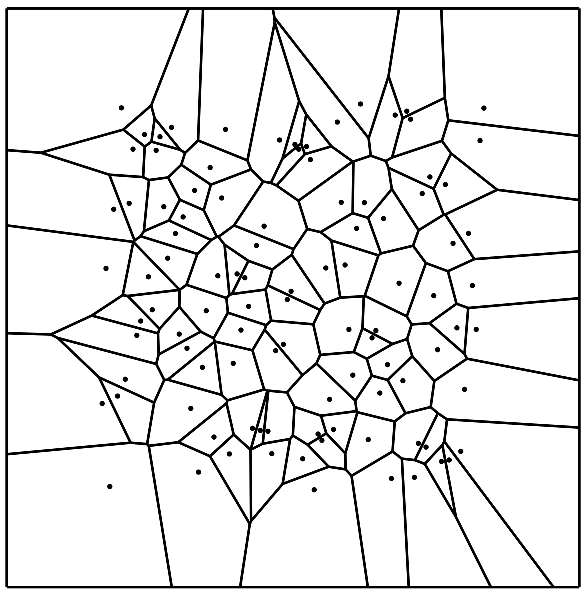

Figure 9: Voronoi diagram of same 100 points

```
  public:
    Vec2 (int, int);
    int x();
    int y();
    void print();
    Vec2 operator + ( Vec2 other );
    bool operator == ( Vec2 other );

  private:
    int a, b;
} Vec2;

Vec2::Vec2 ( int x, int y )
{
  a = x; b = y;
}

Vec2 Vec2::operator + ( Vec2 other )
{
  return Vec2 ( a+other.x(), b+other.y() );
}

bool Vec2::operator == ( Vec2 other )
{
  return ( a == other.x() && b == other.y() );
}

void Vec2::print()
{
  cout << a << ’ ’ << b << endl;
}

int Vec2::x()
{
  return a;
}

int Vec2::y()
{
  return b;
}

int main()
{
  Vec2 a = Vec2 (1,2), b = Vec2 (3,4), c = Vec2 (4,6), d(0,0);
  d = a+b;
  a.print();
  b.print();
  c.print();
  d.print();
  cout << (c == d) << endl;

  return 0;
}
-------------------a.out
1 2
3 4
4 6
4 6
1
```

31.2 Reference and pointers

C++ allows for *reference types* which are different from *pointer types*.

```
  int a, *b, &c;
```

declares `a, b, c` to be integer, pointer to integer, and reference to integer variable. The compiler will not permit `c` to be declared without initialisation. It must be initialised to another variable or array member.

Reference-type variables have been seen in routine arguments.

```
#include <iostream>

using namespace std;

int main()
{
  int x, &y = x, z, a[3], &w = a[1];
        // a and w not used: just showing
        // they are accepted.

  x = 14;
  y = x;
  z = 20;

  cout << x << ' ' << y << ' ' << z << endl;
  ++x;
  cout << x << ' ' << y << ' ' << z << endl;
  ++y;
  cout << x << ' ' << y << ' ' << z << endl;
  y = z;
  cout << y << ' ' << z << ' ' << z << endl;
  ++y;
  cout << y << ' ' << z << ' ' << z << endl;

  return 0;
}
------------------a.out
14 14 20
15 15 20
16 16 20
20 20 20
21 20 20
```

31.3 Routine arguments and this

In a class object, **this** is the *address* of the object. Class member `x` can be accessed as `this->x`. Through `this,` one can access class members whose names have been taken by routine arguments. For example

```
#include <iostream>

using namespace std;

typedef class Complex
{
  public:
    Complex ( double x, double y );
    void print();

  private:
    double x,y;
} Complex;

void Complex::print()
{
  cout << x << " + " << y << " i\n";
}

Complex::Complex ( double x, double y )
{
  this->x = x; this->y = y;
}

int main()
{
  Complex z ( 1 , 2 );
  z.print();
  return 0;
}
--------------a.out
1 + 2 i
```

31.4 Privacy

Question. What's the point of privacy?

Answer. The difference between a pointer-friendly language such as C with its **struct**-based objects and an object-oriented language with its **class**-based objects is that the structures in C are inert, and can be modified anywhere.

In C++ the objects are more like agents, each with its own job to carry out. Even though they are probably created by the same programmer, they are not allowed to interfere with each other. The 'private' sections of the class are something like notes which are made during performance of your job. You rely on the notes being the notes you wrote, not tampered with. Hence the use of private sections in class definitions.

31.5 Const

In C++ `const` means

the item is constant, **sort of**

```
  const double pi = 3.14159;
```

means what it says, but

```
  void print ( const C & x )
```

means that `x` is of type `C`, passed by reference, but that the routine `print` will not (knowingly) change `x`. If `C` occupies a lot of memory, this saves time and space, using call-by-reference, and gives the kind of protection against alteration guaranteed using call-by-value.

This is the most common usage of `const`.

Constants inside classes are peculiar. With the keyword 'static,' they must be initialised as they would be outside classes, and without that keyword they must be initialised in a peculiar way illustrated below.

```
#include <iostream>
//adapted from Ellis Stroustrup p 292
```

```
using namespace std;

        // Illustrating use of ‘this’ to free names
        // for routine arguments.

typedef class B1 { public:  B1(int); int x; } B1;

B1::B1(int x){this->x = x;}

        // Illustrating peculiar notions of constants

typedef class D
{
public:
  D(int);
  const int c;
  static const int x = 29;
} D;

        // Constant c is not to be changed
        // after constructing an object,
        // but its initialisation is required
        // USING ONLY THE STYLE shown

D::D(int a): c(a+4)
{ }

int main()
{

  D d(10);

  cout << d.c << endl << d.x << endl;

  return 0;
}
```

32 Syllabus for May 2014 exam

Anything in the quizzes is fair game for the final exam. Also, pieces of code which occurred in programming assignments could be asked.

- Data types: `char, int, bool, double`.

- Assignment statements, if, while, etcetera.
- Arrays. Address calculation (1 and 2-dimensional arrays). Array initialisation for tables.
- Character strings. Command-line arguments, atoi(), atof().
- Functions and routines. Call by value; how array arguments are passed; call by reference.
- Simulating functions and routines.
- Variables with the same name.
- Random numbers. `RAND_MAX`. Non-random effects of using modular arithmetic to restrict the range, with linear congruential generators. Better method of producing random integers in a restricted range.
- Structures (introduced only as a prelude to classes).
- Classes. Class members. Public and private. Overloading. Operator overloading.
- Evaluating expressions: order of evaluation; mixed-type expressions; casts.
- Simulating recursive routines and functions.
- Files: ifstream, ofstream, opening a file, closing, cin, cout, eof(), getline().
- Standard template library: `string, vector, set, map`. Iterators used for traversing, also in *find()* and *sort()*. Use of `*it` or `it->...` when `it` is an iterator. References and pointers.
- You should know the STL functions and operators mentioned in Section 25.
- IEEE standard: machine-epsilon for single and double-precision. Accuracy of LU factorisation and of summation.