C++ for C Programmers

Victor Eijkhout

2018

Contents

Chapter 1

Introduction

The C++ language is, to first order of approximation, a superset of the C language. Thus, many C programs are also legal C++ programs. C++ offers obvious improvement, such as object-oriented programming. If you already understand C control structures and functions, this booklet will teach you object oriented programming in C syntax.

However, viewing C++ as 'C with classes' does not lead to idiomatic C++ programs. There are in fact a number of C constructs that should not be used. For instance, C++ has ways of dealing with dynamic memory allocation that are both easier to use, and less dangerous in that they take care of de-allocation for you.

Thus, this booklet will also teach what C mechanisms not to use and how to improve your code with their C++ replacements.

Chapter 2

Input/output

2.1 Formatted output

Formatted output

- `cout` uses default formatting
- Possible: pad a number, use limited precision, format as hex/base2, etc
- Many of these output modifiers need

```
#include <iomanip>
```

Normally, output of numbers takes up precisely the space that it needs:

Code:

```
for (int i=1; i<200000000; i*=10)
  cout << "Number: " << i << endl;
cout << endl;
```

Output [io] cunformat:

```
Number: 1
Number: 10
Number: 100
Number: 1000
Number: 10000
Number: 100000
Number: 1000000
Number: 10000000
Number: 100000000
```

Reserve space

You can specify the number of positions, and the output is right aligned in that space by default:

Code:

```
cout << "Width is 6:" << endl;
for (int i=1; i<200000000; i*=10)
  cout << "Number: "
       << setw(6) << i << endl;
cout << endl;
cout << "Width is 6:" << endl;
cout << setw(6) << 1 << 2 << 3 << endl;
cout << endl;
```

Output [io] width:

```
Width is 6:
Number:      1
Number:     10
Number:    100
Number:   1000
Number:  10000
Number: 100000
Number: 1000000
Number: 10000000
Number: 100000000

Width is 6:
     123
```

Padding character

Normally, padding is done with spaces, but you can specify other characters:

Code:

```
#include <iomanip>
using std::setfill;
using std::setw;
//codesnippet formatpad
  /* ... */

int main() {

  //codesnippet formatpad
  /* ... */
  for (int i=1; i<200000000; i*=10)
    cout << "Number: "
         << setfill('.') << setw(6) << i
 << endl;
```

Output [io] formatpad:

```
Number: .....1
Number: ....10
Number: ...100
Number: ..1000
Number: .10000
Number: 100000
Number: 1000000
Number: 10000000
Number: 100000000
```

Note: single quotes denote characters, double quotes denote strings.

Left alignment

Instead of right alignment you can do left:

Code:

```
#include <iomanip>
using std::left;
using std::setfill;
using std::setw;
  /* ... */
  for (int i=1; i<200000000; i*=10)
    cout << "Number: "
         << left << setfill('.') << setw(6)
 << i << endl;
```

Output [io] formatleft:

```
Number: 1.....
Number: 10....
Number: 100...
Number: 1000..
Number: 10000.
Number: 100000
Number: 1000000
Number: 10000000
Number: 100000000
```

Number base

Finally, you can print in different number bases than 10:

Code:

```
#include <iomanip>
using std::setbase;
using std::setfill;
  /* ... */
  cout << setbase(16) << setfill(' ');
  for (int i=0; i<16; i++) {
    for (int j=0; j<16; j++)
      cout << i*16+j << " " ;
    cout << endl;
  }
```

Output [io] format16:

```
0 1 2 3 4 5 6 7 8 9 a b c d e f
10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1
20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2
30 31 32 33 34 35 36 37 38 39 3a 3b 3c 3
40 41 42 43 44 45 46 47 48 49 4a 4b 4c 4
50 51 52 53 54 55 56 57 58 59 5a 5b 5c 5
60 61 62 63 64 65 66 67 68 69 6a 6b 6c 6
70 71 72 73 74 75 76 77 78 79 7a 7b 7c 7
80 81 82 83 84 85 86 87 88 89 8a 8b 8c 8
90 91 92 93 94 95 96 97 98 99 9a 9b 9c 9
a0 a1 a2 a3 a4 a5 a6 a7 a8 a9 aa ab ac a
b0 b1 b2 b3 b4 b5 b6 b7 b8 b9 ba bb bc b
c0 c1 c2 c3 c4 c5 c6 c7 c8 c9 ca cb cc c
d0 d1 d2 d3 d4 d5 d6 d7 d8 d9 da db dc d
e0 e1 e2 e3 e4 e5 e6 e7 e8 e9 ea eb ec e
f0 f1 f2 f3 f4 f5 f6 f7 f8 f9 fa fb fc f
```

Exercise 2.1. Make the above output more nicely formatted:

```
00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f
etc
```

Exercise 2.2. Use integer output to print fixed point numbers aligned on the decimal:

```
  1.345
 23.789
456.1234
```

Use four spaces for both the integer and fractional part.

Hexadecimal

Hex output is useful for pointers (chapter 7):

```
int i;
cout << "address of i, decimal: "
     << (long)&i << endl;
cout << "address if i, hex    : "
     << std::hex << &i << endl;
```

Back to decimal:

```
cout << hex << i << dec << j;
```

2.2 Floating point output

Floating point precision

Use `setprecision` to set the number of digits before and after decimal point:

Code:

```
#include <iomanip>
using std::left;
using std::setfill;
using std::setw;
using std::setprecision;
  /* ... */
  x = 1.234567;
  for (int i=0; i<10; i++) {
    cout << setprecision(4) << x << endl;
    x *= 10;
  }
```

Output [io] formatfloat:

```
1.235
12.35
123.5
1235
1.235e+04
1.235e+05
1.235e+06
1.235e+07
1.235e+08
1.235e+09
```

(Notice the rounding)

Fixed point precision

Fixed precision applies to fractional part:

Code:

```
x = 1.234567;
cout << fixed;
for (int i=0; i<10; i++) {
  cout << setprecision(4) << x << endl;
  x *= 10;
}
```

Output [io] fix:

```
1.2346
12.3457
123.4567
1234.5670
12345.6700
123456.7000
1234567.0000
12345670.0000
123456700.0000
1234567000.0000
```

Aligned fixed point output

Combine width and precision:

Code:

```
  x = 1.234567;
  cout << fixed;
  for (int i=0; i<10; i++) {
    cout << setw(10) << setprecision(4) << x
 << endl;
    x *= 10;
  }
```

Output [io] align:

```
    1.2346
   12.3457
  123.4567
 1234.5670
12345.6700
123456.7000
1234567.0000
12345670.0000
123456700.0000
1234567000.0000
```

Scientific notation

```
cout << "Combine width and precision:" << endl;
x = 1.234567;
cout << scientific;
for (int i=0; i<10; i++) {
  cout << setw(10) << setprecision(4) << x << endl;
  x *= 10;
```

```
}
```

Output

```
Combine width and precision:
1.2346e+00
1.2346e+01
1.2346e+02
1.2346e+03
1.2346e+04
1.2346e+05
1.2346e+06
1.2346e+07
1.2346e+08
1.2346e+09
```

2.3 Saving and restoring settings

```
ios::fmtflags old_settings = cout.flags();

cout.flags(old_settings);

int old_precision = cout.precision();

cout.precision(old_precision);
```

2.4 File output

Text output to file

Streams are general: work the same for console out and file out.

```
#include <fstream>
```

Use:

```
#include <fstream>
using std::ofstream;
  /* ... */
  ofstream file_out;
  file_out.open("fio_example.out");
  /* ... */
  file_out << number << endl;
  file_out.close();
```

Binary output

```
ofstream file_out;
file_out.open
  ("fio_binary.out",ios::binary);
/* ... */
file_out.write( (char*)(&number),4);
```

2.4.1 Output your own classes

You have used statements like:

```
cout << "My value is: " << myvalue << endl;
```

How does this work? The 'double less' is an operator with a left operand that is a stream, and a right operand for which output is defined; the result of this operator is again a stream. Recursively, this means you can chain any number of applications of << together.

If you want to output a class that you wrote yourself, you have to define how the << operator deals with your class.

```
class container {
  /* ... */
  int value() const {
  /* ... */
  };
  /* ... */
ostream &operator<<(ostream &os,const container &i) {
  os << "Container: " << i.value();
  return os;
};
  /* ... */
  container eye(5);
  cout << eye << endl;
```

2.5 Binary output

Code:

```
#include <fstream>
using std::ofstream;
  /* ... */
  ofstream file_out;
  file_out.open("fio_example.out");
  /* ... */
  file_out << number << endl;
  file_out.close();
```

Output [io] fio:

```
echo 24 | ./fio ; \
  cat fio_example.out
A number please:
Written.
24
```

Code:

```
ofstream file_out;
file_out.open
  ("fio_binary.out",ios::binary);
/* ... */
file_out.write( (char*)(&number),4);
```

Output [io] fiobin:

```
echo 25 | ./fiobin ; \
  od fio_binary.out
A number please: Written.
0000000    000031  000000
0000004
```

2.6 Input

Better terminal input

It is better to use `getline`. This returns a string, rather than a value, so you need to convert it with the following bit of magic:

```
#include <iostream>
using std::cin;
using std::cout;
using std::endl;
#include <sstream>
using std::stringstream;
  /* ... */
  std::string saymany;
  int howmany;

  cout << "How many times? ";
  getline( cin,saymany );
  stringstream saidmany(saymany);
  saidmany >> howmany;
```

You can not use `cin` and `getline` in the same program.

More info: http://www.cplusplus.com/forum/articles/6046/.

2.6.1 File input

File input streams

Input file stream, method `open`, then use `getline` to read one line at a time:

```
#include <fstream>
using std::ifstream;
  /* ... */
  ifstream input_file;
  input_file.open("fox.txt");
  string oneline;
  while (input_file) {
    getline(input_file,oneline);
    cout << "Got line: <<" << oneline << ">>" << endl;
  }
```

Exercise 2.3. Put the following text in a file:

```
the quick brown fox
jummps over the
lazy dog.
```

Open the file, read it in, and count how often each letter in the alphabet occurs in it

2.6.2 Input streams

Test, mostly for file streams: `is_eof is_open`

2.7 Sources used in this chapter

Listing of code/io/cunformat:

```
/*****************************************************************
 ****
 **** This file belongs with the course
 **** Introduction to Scientific Programming in C++/Fortran2003
 **** copyright 2017/8 Victor Eijkhout eijkhout@tacc.utexas.edu
 ****
 **** cunformat.cxx : default formatting
 ****
 *****************************************************************/

#include <iostream>
using std::cout;
using std::endl;

int main() {

  //codesnippet cunformat
  for (int i=1; i<200000000; i*=10)
    cout << "Number: " << i << endl;
  cout << endl;
  //codesnippet end

  return 0;
}
```

Listing of code/io/width:

```
/*****************************************************************
 ****
 **** This file belongs with the course
 **** Introduction to Scientific Programming in C++/Fortran2003
 **** copyright 2017/8 Victor Eijkhout eijkhout@tacc.utexas.edu
 ****
 **** width.cxx : setting output width
 ****
 *****************************************************************/

#include <iostream>
```

```
using std::cout;
using std::endl;
#include <iomanip>
using std::right;
using std::setbase;
using std::setfill;
using std::setw;

int main() {

  //codesnippet formatwidth6
  cout << "Width is 6:" << endl;
  for (int i=1; i<200000000; i*=10)
    cout << "Number: "
         << setw(6) << i << endl;
  cout << endl;
  cout << "Width is 6:" << endl;
  cout << setw(6) << 1 << 2 << 3 << endl;
  cout << endl;
  //codesnippet end

  return 0;
}
```

Listing of code/io/formatpad:

```
/****************************************************************
 ****
 **** This file belongs with the course
 **** Introduction to Scientific Programming in C++/Fortran2003
 **** copyright 2017/8 Victor Eijkhout eijkhout@tacc.utexas.edu
 ****
 **** formatpad.cxx : padded io
 ****
 ****************************************************************/

#include <iostream>
using std::cout;
using std::endl;
//codesnippet formatpad
#include <iomanip>
using std::setfill;
using std::setw;
//codesnippet formatpad

int main() {

  //codesnippet formatpad
  for (int i=1; i<200000000; i*=10)
    cout << "Number: "
         << setfill('.') << setw(6) << i
 << endl;
  //codesnippet end

  return 0;
}
```

Listing of code/io/formatleft:

```
/****************************************************************
 ****
 **** This file belongs with the course
 **** Introduction to Scientific Programming in C++/Fortran2003
 **** copyright 2017/8 Victor Eijkhout eijkhout@tacc.utexas.edu
 ****
 **** formatleft.cxx : left aligned io
 ****
 ****************************************************************/

#include <iostream>
using std::cin;
using std::cout;
using std::endl;
//codesnippet formatleft
#include <iomanip>
using std::left;
using std::setfill;
using std::setw;
//codesnippet end

int main() {

  //codesnippet formatleft
  for (int i=1; i<200000000; i*=10)
    cout << "Number: "
         << left << setfill('.') << setw(6)
 << i << endl;
  //codesnippet end
  cout << endl;

  return 0;
}
```

Listing of code/io/format16:

```
/****************************************************************
 ****
 **** This file belongs with the course
 **** Introduction to Scientific Programming in C++/Fortran2003
 **** copyright 2017/8 Victor Eijkhout eijkhout@tacc.utexas.edu
 ****
 **** format16.cxx : base 16 formatted io
 ****
 ****************************************************************/

#include <iostream>
using std::cout;
using std::endl;
//codesnippet format16
#include <iomanip>
using std::setbase;
using std::setfill;
```

```
//codesnippet end

int main() {

  //codesnippet format16
  cout << setbase(16) << setfill(' ');
  for (int i=0; i<16; i++) {
    for (int j=0; j<16; j++)
      cout << i*16+j << " " ;
    cout << endl;
  }
  //codesnippet end

  return 0;
}
```

Listing of code/io/formatfloat:

```
/****************************************************************
 ****
 **** This file belongs with the course
 **** Introduction to Scientific Programming in C++/Fortran2003
 **** copyright 2017/8 Victor Eijkhout eijkhout@tacc.utexas.edu
 ****
 **** formatfloat.cxx : floating point output
 ****
 ****************************************************************/

#include <iostream>
using std::cin;
using std::cout;
using std::endl;
//codesnippet formatfloat
#include <iomanip>
using std::left;
using std::setfill;
using std::setw;
using std::setprecision;
//codesnippet end

int main() {

  float x;
  //codesnippet formatfloat
  x = 1.234567;
  for (int i=0; i<10; i++) {
    cout << setprecision(4) << x << endl;
    x *= 10;
  }
  //codesnippet end

  return 0;
}
```

Listing of code/io/fix:

```
/****************************************************************
 ****
 **** This file belongs with the course
 **** Introduction to Scientific Programming in C++/Fortran2003
 **** copyright 2018 Victor Eijkhout eijkhout@tacc.utexas.edu
 ****
 **** fix.cxx : fixed precision io
 ****
 ****************************************************************/

#include <iostream>
using std::cin;
using std::cout;
using std::endl;

#include <iomanip>
using std::fixed;
using std::setprecision;
using std::setw;

int main() {

  double x;
  //codesnippet fixfrac
  x = 1.234567;
  cout << fixed;
  for (int i=0; i<10; i++) {
    cout << setprecision(4) << x << endl;
    x *= 10;
  }
  //codesnippet end

  return 0;
}
```

Listing of code/io/align:

```
/****************************************************************
 ****
 **** This file belongs with the course
 **** Introduction to Scientific Programming in C++/Fortran2003
 **** copyright 2018 Victor Eijkhout eijkhout@tacc.utexas.edu
 ****
 **** fix.cxx : fixed precision io
 ****
 ****************************************************************/

#include <iostream>
using std::cin;
using std::cout;
using std::endl;

#include <iomanip>
using std::fixed;
using std::setprecision;
using std::setw;
```

```
int main() {

  double x;
  //codesnippet align
  x = 1.234567;
  cout << fixed;
  for (int i=0; i<10; i++) {
    cout << setw(10) << setprecision(4) << x
 << endl;
    x *= 10;
  }
  //codesnippet end

  return 0;
}
```

Listing of code/io/fio:

```
/*****************************************************************
 ****
 **** This file belongs with the course
 **** Introduction to Scientific Programming in C++/Fortran2003
 **** copyright 2016/8 Victor Eijkhout eijkhout@tacc.utexas.edu
 ****
 **** fio.cxx : file io
 ****
 *****************************************************************/

#include <iostream>
using std::cin;
using std::cout;
using std::endl;
#include <iomanip>
using std::right;
using std::setbase;
using std::setfill;
using std::setw;

//codesnippet fio
#include <fstream>
using std::ofstream;
//codesnippet end

int main() {

  //codesnippet fio
  ofstream file_out;
  file_out.open("fio_example.out");
  //codesnippet end

  int number;
  cout << "A number please: ";
  cin >> number;
  cout << endl;
  //codesnippet fio
```

```
  file_out << number << endl;
  file_out.close();
  //codesnippet end
  cout << "Written." << endl;

  return 0;
}
```

Listing of code/io/fiobin:

```
/*****************************************************************
 ****
 **** This file belongs with the course
 **** Introduction to Scientific Programming in C++/Fortran2003
 **** copyright 2017/8 Victor Eijkhout eijkhout@tacc.utexas.edu
 ****
 **** fiobin.cxx : binary file io
 ****
 *****************************************************************/

#include <iostream>
using std::cin;
using std::cout;
using std::endl;
#include <iomanip>
using std::right;
using std::setbase;
using std::setfill;
using std::setw;

#include <fstream>
using std::ofstream;
using std::ios;

int main() {

  //codesnippet fiobin
  ofstream file_out;
  file_out.open
    ("fio_binary.out",ios::binary);
  //codesnippet end

  int number;
  cout << "A number please: ";
  cin >> number;
  //  file_out << number ;
  //codesnippet fiobin
  file_out.write( (char*)(&number),4);
  //codesnippet end
  file_out.close();
  cout << "Written." << endl;

  return 0;
}
```

Chapter 3

Arrays

3.1 Introduction

An *array* is an indexed data structure, that for each index stores an integer, floating point number, character, object, et cetera. In scientific applications, arrays often correspond to vectors and matrices, potentialy of quite large size. (If you know about the Finite Element Method (FEM), you know that vectors can have sizes in the millions or beyond.)

In this chapter you will see the C++ `vector` construct, which implements the notion of an array of things, whether they be numbers, strings, object. While C++ can use the C mechanisms for arrays, for almost all purposes it is better to use `vector`. In particular, this is a safer way to do dynamic allocation. The old mechanisms are briefly discussed in section 2.4.

3.1.1 Initialization

To define an array you need to declare its size, and you need to give it its contents. Those actions don't necessarily have to occur together. (And the contents can later change, as with any other variable; maybe even the size can change.) However, if you know the array size and contents already before you run your code, you can create the whole array in one statement. There is more than one syntax for doing so.

Code:

```
{
  vector<int> numbers{5,6,7,8,9,10};
  cout << numbers[3] << endl;
}
{
  vector<int> numbers = {5,6,7,8,9,10};
  numbers[3] = 21;
  cout << numbers[3] << endl;
}
```

Output [array] dynamicinit:

```
8
21
```

As you see in this example, if `a` is an array, and `i` and integer, then `a[i]` is the i'th element.

- An array element `a[i]` can be used to get the value of an array element, or it can occur in the left-hand side of an assignment to set the value.
- The *array index* (or *array subscript*) `i` starts numbering at zero.
- Therefore, if an array has n elements, its last element has index `n-1`.
- If you try to get an array elements outside the bounds of the array, the compiler will only detect this in simple cases. More likely, your program may give a runtime error, but that does not necessarily happen. You could just get some random value.

3.1.2 Ranging over an array

If you need to consider all the elements in an array, you typically use a `for` loop. There are various ways of doing this.

First of all consider the cases where you consider the array as a collection of elements, and the loop functions like a mathematical 'for all'.

Range over elements

You can write a *range-based for* loop, which considers the elements as a collection.

```
for ( float e : array )
  // statement about element with value e
for ( auto e : array )
  // same, with type deduced by compiler
```

Code:

```
vector<int> numbers = {1,4,2,6,5};
int tmp_max = numbers[0];
for (auto v : numbers)
  if (v>tmp_max)
    tmp_max = v;
cout << "Max: " << tmp_max << " (should be 6)" << endl;
```

Output [array] dynamicmax:

```
Max: 6 (should be 6)
```

(You can spell out the type of the array element, but such type specifications can be complex. In that case, using `auto` is quite convenient.)

If you actually need the array index of the element, you can use a traditional `for` loop with loop variable.

Indexing the elements

You can write an *indexed for* loop, which uses an index variable that ranges from the first to the last element.

```
for (int i= /* from first to last index */ )
  // statement about index i
```

Example: find the maximum element and where it occurs.

Code:

```
int tmp_idx = 0;
int tmp_max = numbers[tmp_idx];
for (int i=0; i<5; i++) {
  int v = numbers[i];
  if (v>tmp_max) {
    tmp_max = v; tmp_idx = i;
  }
}
cout << "Max: " << tmp_max
     << " at index: " << tmp_idx << endl;
```

Output [array] idxmax:

```
Max: 6 at index: 3
```

Exercise 3.1. Find the element with maximum absolute value in an array. Use:
`vector<int> numbers = {1,-4,2,-6,5};`

Which mechanism do you use for traversing the array?
Hint:

```
#include <cmath>
..
absx = abs(x);
```

Exercise 3.2. Find the location of the first negative element in an array.
Which mechanism do you use?

Exercise 3.3. Check whether an array is sorted.

Indexing with pre/post increment

Shorthand:

```
x = a[i++]; /* is */ x = a[i]; i++;
y = b[++i]; /* is */ i++; y = b[i];
```

Range over elements by reference

Range-based loop indexing makes a copy of the array element. If you want to alter the array, use a reference:

Code:

```
vector<int> numbers = {1,4,2,6,5};
for ( auto &v : numbers )
  v *= 3;
cout << "Scale 0'th by 3: " << numbers[0] << endl;
```

Output [array] dynamicscale:

```
Scale 0'th by 3: 3
```

Exercise 3.4. If you do the prime numbers project, you can now do exercise **??**.

3.1.3 Vector are a class

You wouldn't tell it from the above examples, but vectors actually form a `vector` class. The angle bracket notation means that we have a class that is parametrized with the type (see chapter 9 for the details), and you can have vectors of ints, vectors of chars, et cetera. We can now say that `vector<int>` is a type, pronounced 'vector-of-int', and you can make new variables of that type.

What we were doing above was creating an object and initializing it in one go. Let's decouple these actions.

Vector definition

Definition, mostly without initialization.

```
#include <vector>
using std::vector;

vector<type> name;
vector<type> name(size);
```

where

- `vector` is a keyword,
- `type` (in angle brackets) is any elementary type or class name,
- `name` is up to you, and
- `size` is the (initial size of the array). This is an integer, or more precisely, a `size_t` parameter.

Remark 1 *There is also an* `array` *class, which at first glance looks like a non-resizable variant of* `vector`*. However, it is limited to arrays where the size is known at compile time.*

3.1.3.1 Vector initialization

Vector initialization

You can initialize a vector as a whole:

```
vector<int> odd_array{1,3,5,7,9};
vector<int> even_array = {0,2,4,6,8};
```

(This syntax requires compilation with the `-std=c++11` option.)

Vector initialization'

There is a syntax for initializing a vector with a constant:

```
vector<float> x(25,3.15);
```

which gives a vector of size 25, with all elements initialized to 3.15.

3.1.3.2 Element access

Accessing vector elements

You have already seen the square bracket notation:

```
vector<double> x(5, 0.1 );
x[1] = 3.14;
cout << x[2];
```

Alteratively:

```
x.at(1) = 3.14;
cout << x.at(2);
```

Safer, slower.

Ranging over a vector

```
for ( auto e : my_vector)
  cout << e;
```

Note that `e` is a copy of the vector element:

Code:

```
vector<float> myvector
  = {1.1, 2.2, 3.3};
for ( auto e : myvector )
  e *= 2;
cout << myvector[2] << endl;
```

Output [array] vectorrangecopy:

```
3.3
```

Ranging over a vector by reference

To set array elements, make `e` a reference:

```
for ( auto &e : my_vector)
  e = ....
```

Code:

```
vector<float> myvector
  = {1.1, 2.2, 3.3};
for ( auto &e : myvector )
  e *= 2;
cout << myvector[2] << endl;
```

Output [array] vectorrangeref:

```
6.6
```

Vector copy

Vectors can be copied just like other datatypes:

Code:

```
vector<float> v(5,0), vcopy;
v[2] = 3.5;
vcopy = v;
cout << vcopy[2] << endl;
```

Output [array] vectorcopy:

```
./vectorcopy
3.5
```

3.1.3.3 Vector methods

A `vector` is an object. Let's take a look at some of the methods that are defined for it.

First of all, there is a way of accessing vector elements through the `at` method. It has the big advantage that it does *bounds checking*.

Vector indexing

Your choice: fast but unsafe, or slower but safe

```
vector<double> x(5);
x[5] = 1.; // will probably work
x.at(5) = 1.; // runtime error!
```

Safer, but also slower; see below.

The method `size` gives the size of the vector:

```
vector<char> words(37);
cout << words.size(); // will print 37
```

The existence of this method means that you don't have to remember the size of a vector: it has that information internally.

Exercise 3.5. Create a `vector` x of `float` elements, and set them to random values.
Now normalize the vector in L_2 norm, that is, scale each element by the same coefficient α so that $\sum_i x_i^2 = 1$. Check the correctness of your calculation.

3.1.4 Vectors are dynamic

There is an important difference between vectors and arrays: a vector can be grown or shrunk after its creation. Use the `push_back` method to add elements at the end:

Dynamic extension

Extend with `push_back`:

Code:

```
vector<int> array(5,2);
array.push_back(35);
cout << array.size() << endl;
cout << array[array.size()-1] << endl;
```

Output [array] vectorend:

```
6
35
```

also `pop_back`, `insert`, `erase`.
Flexibility comes with a price.

This is not a good way of creating arrays: if you know the size, it is better to reserve the vector at the correct size:

```
vector<int> iarray;
iarray.reserve(100);
for ( ... )
  iarray.push_back( ... );
```

Other methods that change the size: `insert`, `erase`.

3.1.5 Vectors and functions

3.1.5.1 Pass vector to function

Vector as function argument

You can pass a vector to a function:

```
void print0( vector<double> v ) {
  cout << v[0] << endl;
};
```

Vectors, like any argument, are passed by value, so the vector is actually copied into the function.

Vector pass by value example

Code:

```
void set0
  ( vector<float> v,float x )
{
  v[0] = x;
}
  /* ... */
  vector<float> v(1);
  v[0] = 3.5;
  set0(v,4.6);
  cout << v[0] << endl;
```

Output [array] vectorpassnot:

```
./vectorpassnot
3.5
```

3.1.5.2 Vector as function return

Vector as function return

You can have a vector as return type of a function:

Code:

```
vector<int> make_vector(int n) {
  vector<int> x(n);
  x[0] = n;
  return x;
}
  /* ... */
  vector<int> x1 = make_vector(10); // "auto" also possible!
  cout << "x1 size: " << x1.size() << endl;
  cout << "zero element check: " << x1[0] << endl;
```

Output [array] vectorreturn:

```
./vectorreturn
x1 size: 10
zero element check: 10
```

Exercise 3.6. Write code to take a vector of integers, and construct two vectors, one containing all the odd inputs, and one containing all the even inputs. So:

```
input:
   5,6,2,4,5
output:
   5,5
   6,2,4
```

Can you write a function that accepts a vector and returns two vectors with the above functionality.

Vector pass by reference

If you want to alter the vector, you have to pass by reference:

Code:

```
void set0
  ( vector<float> &v,float x )
{
  v[0] = x;
}
  /* ... */
  vector<float> v(1);
  v[0] = 3.5;
  set0(v,4.6);
  cout << v[0] << endl;
```

Output [array] vectorpassref:

```
./vectorpassref
4.6
```

Exercise 3.7. Write functions `random_vector` and `sort` to make the following main program work:

```
int length = 10;
vector<float> values = random_vector(length);
sort(values);
```

(This creates a vector of random values of a specified length, and then sorts it.)
See section **??** for the random fuction.

3.1.6 Vectors in classes

You may want an object that contains a vector, where the size of the vector is passed to the constructor. Since the class definition is an abstract definition of all the object, clearly you can not have the array size there.

```
class witharray {
private:
  vector<int> the_array( ???? );
public:
  witharray( int n ) {
    thearray( ???? n ???? );
  }
}
```

The solution is to specify a vector without size in the class definition, which creates a vector of size zero. When you create an object, you then create a vector of the right size, and write that over the vector member of the object.

Create and assign

The following mechanism works:

```
class witharray {
private:
  vector<int> the_array;
public:
  witharray( int n ) {
    thearray = vector<int>(n);
  }
}
```

You could read this as

- `vector<int> the_array` declares a int-vector variable, and
- `thearray = vector<int>(n)` assigns an array to it.

However, technically, it actually does the following:

- The class object initially has a zero-size vector;
- the expression `vector<int>(n)` creates an anonymous vector of size `n`;
- which is then assigned to the variable `the_array`,
- so now you have an object with a vector of size `n` internally.

3.1.7 Dynamic size of vector

It is tempting to use `push_back` to create a vector dynamically.

Dynamic size extending

```
vector<int> iarray;
```

creates a vector of size zero. You can then

```
iarray.push_back(5);
iarray.push_back(32);
iarray.push_back(4);
```

However, this dynamic resizing involves memory management, and maybe operating system functions. This will probably be inefficient. Therefore you should use such dynamic mechanisms only when strictly necessary.

3.1.8 Iterators

You have seen how you can iterate over a vector

- by a for loop over the indices, and
- with a range-based loop over the indices.

There is a third way, which is actually the basic mechanism underlying the range-based looping.

An *iterator* is a pointer to a vector element. Mirroring the index-loop convention of

```
for (int i=0; i<hi; i++)
  element = vec.at(i);
```

you can iterate:

```
for (auto elt_ptr=vec.begin(); elt_ptr<vec.end(); ++elt_ptr)
  element = *elt_ptr;
```

Some remarks:

- This is one of the very few places where you need the asterisk in C++ for *dereferencing*; section **??**.
- As with a normal loop, the `end` iterator point just beyond the end of the vector.
- You can do *pointer arithmetic* on iterators, as you can see in the `++elt_ptr` update part of the loop header.

Another illustration of pointer arithmetic on iterators is getting the last element of a vector:

Code:

```
vector<int> array(5,2);
array.push_back(35);
cout << array.size() << endl;
cout << array[array.size()-1] << endl;
```

Output [array] vectorend:

```
6
35
```

Code:

```
vector<int> array(5,2);
array.push_back(35);
cout << array.size() << endl;
cout << array[array.size()-1] << endl;
cout << *( --array.end() ) << endl;
```

Output [array]
vectorenditerator:

3.1.9 Timing

Different ways of acessing a vector can have drastically different timing cost.

Vector extension

You can push elements into a vector:

```
  vector<int> flex;
/* ... */
  for (int i=0; i<LENGTH; i++)
    flex.push_back(i);
```

If you allocate the vector statically, you can assign with `at`:

```
  vector<int> stat(LENGTH);
/* ... */
  for (int i=0; i<LENGTH; i++)
    stat.at(i) = i;
```

Vector extension

With subscript:

```
  vector<int> stat(LENGTH);
/* ... */
  for (int i=0; i<LENGTH; i++)
    stat[i] = i;
```

You can also use `new` to allocate (see section **??**):

```
  int *stat = new int[LENGTH];
/* ... */
  for (int i=0; i<LENGTH; i++)
    stat[i] = i;
```

Timings are partly predictable, partly surprising:

Timing

```
Flexible time: 2.445
Static at time: 1.177
Static assign time: 0.334
Static assign time to new: 0.467
```

The increased time for `new` is a mystery.

So do you use `at` for safety or `[]` for speed? Well, you could use `at` during development of the code, and insert

```
#define at(x) operator[](x)
```

for production.

3.2 Wrapping a vector in an object

You may want to a create objects that contain a vector, for instance because you want to add some methods.

```
class printable {
private:
  vector<int> values;
public:
  printable(int n) {
    values = vector<int>(n);
  };
  string stringed() {
    string p("");
    for (int i=0; i<values.size(); i++)
      p += to_string(values[i])+" ";
    return p;
  };
};
```

Unfortunately this means you may have to recreate some methods:

```
int &at(int i) {
  return values.at(i);
};
```

3.3 Multi-dimensional cases

3.3.1 Matrix as vector of vectors

Multi-dimensional vectors

Multi-dimensional is harder with vectors:

```
vector<float> row(20);
vector<vector<float>> rows(10,row);
```

Vector of vectors.

This is not the best implementation of a matrix, for instance because the elements are not contiguous. However, let's continue with it for a moment.

Matrix class

```
class matrix {
private:
  int rows,cols;
  vector<vector<double>> elements;
public:
  matrix(int m,int n) {
    rows = m; cols = n;
    elements =
      vector<vector<double>>(m,vector<double>(n));
  }
  void set(int i,int j,double v) {
    elements.at(i).at(j) = v;
  };
  double get(int i,int j) {
    return elements.at(i).at(j);
  };
};
```

Matrix class'

Better idea:

```
  elements = vector<double>(rows*cols);
  ...
  void get(int i,int j) {
    return elements.at(i*cols+j);
  }
```

Exercise 3.8. Add methods such as `transpose`, `scale` to your matrix class.
Implement matrix-matrix multiplication.

3.3.2 A better matrix class

You can make a 'pretend' matrix by storing a long enough `vector` in an object:

```
class matrix {
private:
  std::vector<double> the_matrix;
  int m,n;
public:
  matrix(int m,int n) {
    this->m = m; this->n = n;
```

```
    the_matrix.reserve(m*n);
  };
  void set(int i,int j,double v) {
    the_matrix[ i*n +j ] = v;
  };
  double get(int i,int j) {
    return the_matrix[ i*n +j ];
  };
  /* ... */
};
```

The most important advantage of this is that it is compatible with how many libraries and codes store a matrix traditionally.

The syntax for `set` and `get` can be improved.

Exercise 3.9. Write a method `element` of type `double&`, so that you can write
`A.element(2,3) = 7.24;`

3.4 Static arrays

For small arrays you can use a different syntax.

Code:

```
{
  int numbers[] = {5,4,3,2,1};
  cout << numbers[3] << endl;
}
{
  int numbers[5]{5,4,3,2,1};
  numbers[3] = 21;
  cout << numbers[3] << endl;
}
```

Output [array] staticinit:

```
2
21
```

This has the (minimal) advantage of not having the overhead of a class mechanism. On the other hand, it has a number of disadvantages:

- You can not query the size of an array by its name: you have to store that information separately in a variable.
- Passing such an array to a function is really passing the address of its first element, so it is always (sort of) by reference.

Range-based indexing works the same as with vectors:

Code:

```
int numbers[] = {1,4,2,6,5};
int tmp_max = numbers[0];
for (auto v : numbers)
  if (v>tmp_max)
    tmp_max = v;
cout << "Max: " << tmp_max << " (should be 6)" << endl;
```

Output [array] rangemax:

```
Max: 6 (should be 6)
```

3.5 Old-style arrays

Static arrays are really an abuse of the equivalence of arrays and addresses of the C programming language. This appears for instance in parameter passing mechanisms.

3.5.1 C-style arrays and subprograms

Arrays can be passed to a subprogram, but the bound is unknown there.

```
void array_set( double ar[],int idx,double val) {
  ar[idx] = val;
}
  array_set(array,1,3.5);
```

Exercise 3.10. Rewrite the above exercises where the sorting tester or the maximum finder is in a subprogram.

Unlike with scalar arguments, array arguments can be altered by a subprogram: it is as if the array is always passed by reference. This is not strictly true: what happens is that the address of the first element of the array is passed. Thus we are really dealing with pass by value, but it is the array address that is passed rather than its value.

3.5.2 Multi-dimensional arrays

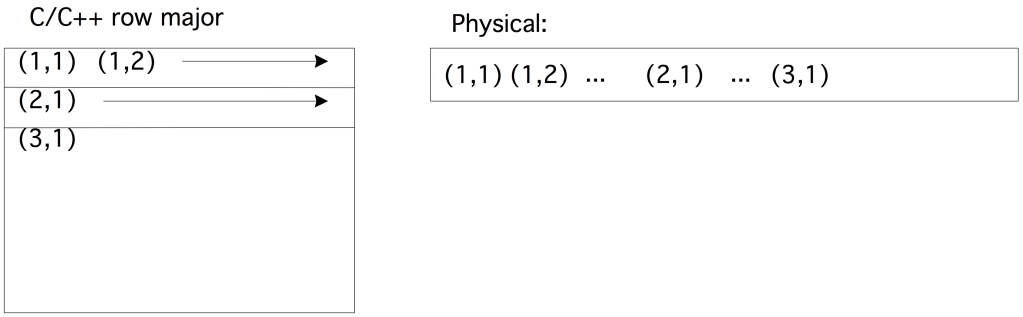
Multi-dimensional arrays can be declared and used with a simple extension of the prior syntax:

```
float matrix[15][25];

for (int i=0; i<15; i++)
  for (int j=0; j<25; j++)
    // something with matrix[i][j]
```

Passing a multi-dimensional array to a function, only the first dimension can be left unspecified:

```
void print12( int ar[][6] ) {
  cout << "Array[1][2]: " << ar[1][2] << endl;
  return;
}
    int array[5][6];
    array[1][2] = 3;
    print12(array);
```



3.5.3 Memory layout

Puzzling aspects of arrays, such as which dimensions need to be specified and which not in a function call, can be understood by considering how arrays are stored in memory. The question then is how a two-dimensional (or higher dimensional) array is mapped to memory, which is linear.

- A one-dimensional array is stored in contiguous memory.
- A two-dimensional array is also stored contiguously, with first the first row, then the second, et cetera.
- Higher dimensional arrays continue this notion, with contiguous blocks of the highest so many dimensions.

As a result of this, indexing beyond the end of a row, brings you to the start of the next row:

```
void print06( int ar[][6] ) {
  cout << "Array[0][6]: " << ar[0][6] << endl;
  return;
}
    int array[5][6];
    array[1][0] = 35;
    print06(array);
```

We can now also understand how arrays are passed to functions:

- The only information passed to a function is the address of the first element of the array;
- In order to be able to find location of the second row (and third, et cetera), the subprogram needs to know the length of each row.
- In the higher dimensional case, the subprogram needs to know the size of all dimensions except for the first one.

3.6 Exercises

Exercise 3.11. Program *bubble sort*: go through the array comparing successive pairs of elements, and swapping them if the second is smaller than the first. After you have gone through the array, the largest element is in the last location. Go through the array again, swapping elements, which puts the second largest element in the one-before-last location. Et cetera.

Pascal's triangle

Pascal's triangle contains binomial coefficients:

```
Row    1:                      1
Row    2:                    1   1
Row    3:                  1   2   1
Row    4:                1   3   3   1
Row    5:              1   4   6   4   1
Row    6:            1   5  10  10   5   1
Row    7:          1   6  15  20  15   6   1
Row    8:        1   7  21  35  35  21   7   1
Row    9:      1   8  28  56  70  56  28   8   1
Row   10:    1   9  36  84 126 126  84  36   9   1
```

where

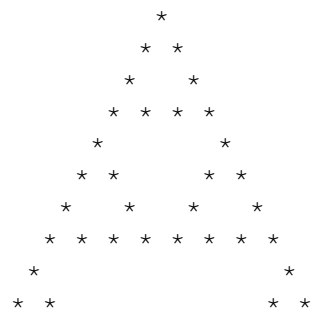
$$p_{rc} = \binom{r}{c} = \frac{r!}{c!(r-c)!}.$$

The coefficients can easily be computed from the recurrence

$$p_{rc} = \begin{cases} 1 & c \equiv 1 \vee c \equiv r \\ p_{r-1,c-1} + p_{r-1,c} \end{cases}$$

Exercise 3.12.

- Write a class `pascal` so that `pascal(n)` is the object containing n rows of the above coefficients. Write a method `get(i,j)` that returns the (i, j) coefficient.
- Write a method `print` that prints the above display.
- Write a method `print(int m)` that prints a star if the coefficient modulo m is nonzero, and a space otherwise.



- The object needs to have an array internally. The easiest solution is to make an array of size $n \times n$.
  Bonus: when you have that code working, optimize your code to use precisely enough space for the coefficients.

Exercise 3.13. A knight on the chess board moves by going two steps horizontally or vertically, and one step either way in the orthogonal direction. Given a starting position, find a sequence of moves that brings a knight back to its starting position. Are there starting positions for which such a cycle doesn't exist?

Exercise 3.14. Put eight queens on a chessboard so that none threatens any other.

Exercise 3.15. From the 'Keeping it REAL' book, exercise 3.6 about Markov chains.

3.7 Sources used in this chapter

Listing of code/array/dynamicinit:

```
/****************************************************************
 ****
 **** This file belongs with the course
 **** Introduction to Scientific Programming in C++/Fortran2003
 **** copyright 2018 Victor Eijkhout eijkhout@tacc.utexas.edu
 ****
 **** dynamicinit.cxx : initialization of vectors
 ****
 ****************************************************************/

#include <iostream>
using std::cout;
using std::endl;

#include <vector>
using std::vector;

int main() {

  //codesnippet dynamicinit
  {
    vector<int> numbers{5,6,7,8,9,10};
    cout << numbers[3] << endl;
  }
  {
    vector<int> numbers = {5,6,7,8,9,10};
    numbers[3] = 21;
    cout << numbers[3] << endl;
  }
  //codesnippet end


  return 0;
}
```

Listing of code/array/dynamicmax:

```
/****************************************************************
 ****
 **** This file belongs with the course
 **** Introduction to Scientific Programming in C++/Fortran2003
 **** copyright 2017/8 Victor Eijkhout eijkhout@tacc.utexas.edu
 ***
 **** dynamicmax.cxx : static array length examples
 ****
 ****************************************************************/

#include <iostream>
using std::cin;
using std::cout;
using std::endl;

#include <vector>
using std::vector;
```

```
int main() {

  //examplesnippet dynamicmax
  vector<int> numbers = {1,4,2,6,5};
  int tmp_max = numbers[0];
  for (auto v : numbers)
    if (v>tmp_max)
      tmp_max = v;
  cout << "Max: " << tmp_max << " (should be 6)" << endl;
  //examplesnippet end

  return 0;
}
```

Listing of code/array/idxmax:

```
/*****************************************************************
 ****
 **** This file belongs with the course
 **** Introduction to Scientific Programming in C++/Fortran2003
 **** copyright 2017/8 Victor Eijkhout eijkhout@tacc.utexas.edu
 ****
 **** idxmax.cxx : static array length examples
 ****
 *****************************************************************/

#include <iostream>
using std::cin;
using std::cout;
using std::endl;

#include <vector>
using std::vector;

int main() {

  int numbers[] = {1,4,2,6,5};
  //examplesnippet idxmax
  int tmp_idx = 0;
  int tmp_max = numbers[tmp_idx];
  for (int i=0; i<5; i++) {
    int v = numbers[i];
    if (v>tmp_max) {
      tmp_max = v; tmp_idx = i;
    }
  }
  cout << "Max: " << tmp_max
       << " at index: " << tmp_idx << endl;
  //examplesnippet end

  return 0;
}
```

Listing of code/array/dynamicscale:

```
/*****************************************************************
 ****
```

```
 **** This file belongs with the course
 **** Introduction to Scientific Programming in C++/Fortran2003
 **** copyright 2017/8 Victor Eijkhout eijkhout@tacc.utexas.edu
 ****
 **** dynamicscale.cxx : static array length examples
 ****
 ****************************************************************/

#include <iostream>
using std::cin;
using std::cout;
using std::endl;

#include <vector>
using std::vector;

int main() {

  //examplesnippet dynamicscale
  vector<int> numbers = {1,4,2,6,5};
  for ( auto &v : numbers )
    v *= 3;
  cout << "Scale 0'th by 3: " << numbers[0] << endl;
  //examplesnippet end

  return 0;
}
```

Listing of code/array/vectorrangecopy:

```
/****************************************************************
 ****
 **** This file belongs with the course
 **** Introduction to Scientific Programming in C++/Fortran2003
 **** copyright 2016-8 Victor Eijkhout eijkhout@tacc.utexas.edu
 ****
 **** vectorrangecopy.cxx : range-based indexing over a vector
 ****
 ****************************************************************/

#include <iostream>
using std::cin;
using std::cout;
using std::endl;

#include <vector>
using std::vector;

int main() {

  //codesnippet vectorrangecopy
  vector<float> myvector
    = {1.1, 2.2, 3.3};
  for ( auto e : myvector )
    e *= 2;
  cout << myvector[2] << endl;
  //codesnippet end
```

```
  return 0;
}
```

Listing of code/array/vectorrangeref:

```
/****************************************************************
 ****
 **** This file belongs with the course
 **** Introduction to Scientific Programming in C++/Fortran2003
 **** copyright 2016-8 Victor Eijkhout eijkhout@tacc.utexas.edu
 ****
 **** vectorrangeref.cxx : range-based indexing by reference
 ****
 ****************************************************************/

#include <iostream>
using std::cin;
using std::cout;
using std::endl;

#include <vector>
using std::vector;

int main() {

  //codesnippet vectorrangeref
  vector<float> myvector
    = {1.1, 2.2, 3.3};
  for ( auto &e : myvector )
    e *= 2;
  cout << myvector[2] << endl;
  //codesnippet end

  return 0;
}
```

Listing of code/array/vectorcopy:

```
/****************************************************************
 ****
 **** This file belongs with the course
 **** Introduction to Scientific Programming in C++/Fortran2003
 **** copyright 2016/7 Victor Eijkhout eijkhout@tacc.utexas.edu
 ****
 **** vectorcopy.cxx : example of vector copying
 ****
 ****************************************************************/

#include <iostream>
using std::cin;
using std::cout;
using std::endl;

#include <vector>
using std::vector;

int main() {
```

```
  //codesnippet vectorcopy
  vector<float> v(5,0), vcopy;
  v[2] = 3.5;
  vcopy = v;
  cout << vcopy[2] << endl;
  //codesnippet end

  return 0;
}
```

Listing of code/array/vectorend:

```
/****************************************************************
 ****
 **** This file belongs with the course
 **** Introduction to Scientific Programming in C++/Fortran2003
 **** copyright 2018 Victor Eijkhout eijkhout@tacc.utexas.edu
 ****
 **** vectorend.cxx : example of vector end iterator
 ****
 ****************************************************************/

#include <iostream>
using std::cin;
using std::cout;
using std::endl;

#include <vector>
using std::vector;

int main() {

  cout << "End Bracket" << endl;
  {
    //codesnippet vectorpush
    vector<int> array(5,2);
    array.push_back(35);
    cout << array.size() << endl;
    cout << array[array.size()-1] << endl;
    //codesnippet end
  }
  cout << "... bracket" << endl;

  cout << "End Iterator" << endl;
  {
    //codesnippet vectorpushiterator
    vector<int> array(5,2);
    array.push_back(35);
    cout << array.size() << endl;
    cout << array[array.size()-1] << endl;
    cout << *( --array.end() ) << endl;
    //codesnippet end
  }
  cout << "... iterator" << endl;

  return 0;
```

```
}
```

Listing of code/array/vectorpassnot:

```
/*****************************************************************
 ****
 **** This file belongs with the course
 **** Introduction to Scientific Programming in C++/Fortran2003
 **** copyright 2016/7 Victor Eijkhout eijkhout@tacc.utexas.edu
 ****
 **** vectorpassnot.cxx : example of vector passed by value
 ****
 *****************************************************************/

#include <iostream>
using std::cin;
using std::cout;
using std::endl;

#include <vector>
using std::vector;

//codesnippet vectorpassval
void set0
  ( vector<float> v,float x )
{
  v[0] = x;
}
//codesnippet end

int main() {

  //codesnippet vectorpassval
  vector<float> v(1);
  v[0] = 3.5;
  set0(v,4.6);
  cout << v[0] << endl;
  //codesnippet end

  return 0;
}
```

Listing of code/array/vectorreturn:

```
/*****************************************************************
 ****
 **** This file belongs with the course
 **** Introduction to Scientific Programming in C++/Fortran2003
 **** copyright 2016/7 Victor Eijkhout eijkhout@tacc.utexas.edu
 ****
 **** vectorreturn.cxx : return vector from function
 ****
 *****************************************************************/

#include <iostream>
using std::cin;
using std::cout;
using std::endl;
```

```
#include <vector>
using std::vector;

//codesnippet vectorreturn
vector<int> make_vector(int n) {
  vector<int> x(n);
  x[0] = n;
  return x;
}
//codesnippet end

int main() {

  //codesnippet vectorreturn
  vector<int> x1 = make_vector(10); // "auto" also possible!
  cout << "x1 size: " << x1.size() << endl;
  cout << "zero element check: " << x1[0] << endl;
  //codesnippet end

  return 0;
}
```

Listing of code/array/vectorpassref:

```
/*****************************************************************
 ****
 **** This file belongs with the course
 **** Introduction to Scientific Programming in C++/Fortran2003
 **** copyright 2016/7 Victor Eijkhout eijkhout@tacc.utexas.edu
 ****
 **** vectorpassref.cxx : example of vector passed by reference
 ****
 *****************************************************************/

#include <iostream>
using std::cin;
using std::cout;
using std::endl;

#include <vector>
using std::vector;

//codesnippet vectorpassref
void set0
  ( vector<float> &v,float x )
{
  v[0] = x;
}
//codesnippet end

int main() {

  //codesnippet vectorpassref
  vector<float> v(1);
  v[0] = 3.5;
  set0(v,4.6);
```

```
  cout << v[0] << endl;
  //codesnippet end

  return 0;
}
```

Listing of code/array/vectorend:

```
/****************************************************************
 ****
 **** This file belongs with the course
 **** Introduction to Scientific Programming in C++/Fortran2003
 **** copyright 2018 Victor Eijkhout eijkhout@tacc.utexas.edu
 ****
 **** vectorend.cxx : example of vector end iterator
 ****
 ****************************************************************/

#include <iostream>
using std::cin;
using std::cout;
using std::endl;

#include <vector>
using std::vector;

int main() {

  cout << "End Bracket" << endl;
  {
    //codesnippet vectorpush
    vector<int> array(5,2);
    array.push_back(35);
    cout << array.size() << endl;
    cout << array[array.size()-1] << endl;
    //codesnippet end
  }
  cout << "... bracket" << endl;

  cout << "End Iterator" << endl;
  {
    //codesnippet vectorpushiterator
    vector<int> array(5,2);
    array.push_back(35);
    cout << array.size() << endl;
    cout << array[array.size()-1] << endl;
    cout << *( --array.end() ) << endl;
    //codesnippet end
  }
  cout << "... iterator" << endl;

  return 0;
}
```

Listing of code/array/vectorenditerator:

Listing of code/array/staticinit:

```
/****************************************************************
 ****
 **** This file belongs with the course
 **** Introduction to Scientific Programming in C++/Fortran2003
 **** copyright 2018 Victor Eijkhout eijkhout@tacc.utexas.edu
 ****
 **** staticinit.cxx : initialization of static arrays
 ****
 ****************************************************************/

#include <iostream>
using std::cout;
using std::endl;

#include <vector>
using std::vector;

int main() {

  //codesnippet arrayinit
  {
    int numbers[] = {5,4,3,2,1};
    cout << numbers[3] << endl;
  }
  {
    int numbers[5]{5,4,3,2,1};
    numbers[3] = 21;
    cout << numbers[3] << endl;
  }
  //codesnippet end


  return 0;
}
```

Listing of code/array/rangemax:

```
/****************************************************************
 ****
 **** This file belongs with the course
 **** Introduction to Scientific Programming in C++/Fortran2003
 **** copyright 2017/8 Victor Eijkhout eijkhout@tacc.utexas.edu
 ***
 **** rangemax.cxx : static array length examples
 ****
 ****************************************************************/

#include <iostream>
using std::cin;
using std::cout;
using std::endl;

#include <vector>
using std::vector;

int main() {
```

```
  //examplesnippet rangemax
  int numbers[] = {1,4,2,6,5};
  int tmp_max = numbers[0];
  for (auto v : numbers)
    if (v>tmp_max)
      tmp_max = v;
  cout << "Max: " << tmp_max << " (should be 6)" << endl;
  //examplesnippet end

  return 0;
}
```

Chapter 4

Strings

4.1 Characters

Characters and ints

- Type `char`;
- represents '7-bit ASCII': printable and (some) unprintable characters.
- Single quotes: `char c = 'a'`
- Equivalent to (short) integer: `'x'-'a'` is distance `a--x`

Exercise 4.1. Write a program that accepts an integer $0 \cdots 26$ and prints the so-manieth letter of the alphabet.
Extend your program so that if the input is negative, it prints the minus-so-manieth uppercase letter of the alphabet.

4.2 Basic string stuff

String declaration

```
#include <string>
using std::string;

// .. and now you can use 'string'
```

(Do not use the C legacy mechanisms.)

String creation

A *string* variable contains a string of characters.

```
string txt;
```

You can initialize the string variable (use `-std=c++11`), or assign it dynamically:

```
string txt{"this is text"};
string moretxt("this is also text");
txt = "and now it is another text";
```

Concatenation

Strings can be *concatenated*:

```
txt = txt1+txt2;
txt += txt3;
```

String is like vector

You can query the *size*:

```
int txtlen = txt.size();
```

or use subscripts:

```
cout << "The second character is <<" <<
        txt[1] << ">>" << endl;
```

Iterating over a string

```
for ( auto c : some_string)
  // do something with the character 'c'
```

Exercise 4.2. The oldest method of writing secret messages is the 'Caesar cypher'. You would take an integer s and rotate every character over that many positions:

$s \equiv 3$: "acdz" $\Rightarrow$ "dfgc".

Write a program that accepts an integer and a string, and display the original string rotated over that many positions.

More vector methods

Other methods for the vector class apply: `insert`, `empty`, `erase`, `push_back`, et cetera.

Methods only for `string`: `find` and such.

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

Exercise 4.3. Write a function to print out the digits of a number: `156` should print `one five six`. Use a vector or array of strings, containing the names of the digits.
Start by writing a program that reads a single digit and prints its name.
For the full program it is easiest to generate the digits last-to-first. Then figure out how to print them reversed.

Exercise 4.4. Write a function to convert an integer to a string: the input `205` should give `two hundred fifteen`, et cetera.

Exercise 4.5. Write a pattern matcher, where a period `.` matches any one character, and `x*` matches any number of 'x' characters.
For example:

- The string `abc` matches `a.c` but `abbc` doesn't.
- The string `abbc` matches `ab*c`, as does `ac`, but `abzbc` doesn't.

4.3 Conversion

`to_string`

4.4 C strings

In C a string is essentially an array of characters. C arrays don't store their length, but strings do have functions that implicitly or explicitly rely on this knowledge, so they have a terminator character: ASCII `NULL`. C strings are called *null-terminated* for this reason.

Chapter 5

Parameter passing

C++ functions resemble mathematical functions: you have seen that a function can have an input and an output. In fact, they can have multiple inputs, separated by commas, but they have only one output.

We start by studying functions that look like these mathematical functions. They involve a *parameter passing* mechanism called *passing by value*. Later we will then look at *passing by reference*.

5.0.1 Pass by value

The following style of programming is very much inspired by mathematical functions, and is known as *functional programming*[1].

- A function has one result, which is returned through a return statement. The function call then looks like

```
y = f(x1,x2,x3);
```

- The definition of the C++ parameter passing mechanism says that input arguments are copied to the function, meaning that they don't change in the calling program:

Code:

```
double squared( double x ) {
  x = x*x;
  return x;
}
  /* ... */
  number = 5.1;
  cout << "Input starts as: "
       << number << endl;
  other = squared(number);
  cout << "Input var is now: "
       << number << endl;
  cout << "Output var is: "
       << other << endl;
```

Output [func] passvalue:

```
Input starts as: 5.1
Input var is now: 5.1
Output var is: 26.01
```

We say that the input argument is *passed by value*: its value is copied into the function. In this example, the function parameter `x` acts as a local variable in the function, and it is initialized with a copy of the value of `number` in the main program.

1. There is more to functional programming. For instance, strictly speaking your whole program needs to be based on function calling; there is no other code than function definitions and calls.

Later we will worry about the cost of this copying.

Passing a variable to a routine passes the value; in the routine, the variable is local. So, in this example the value of the argument is not changed:

Code:

```
void change_scalar(int i) { i += 1; }
  /* ... */
  number = 3;
  cout << "Number is 3: " << number << endl;
  change_scalar(number);
  cout << "is it still 3? Let's see: " << number << endl;
```

Output [func] localparm:

```
Number is 3: 3
is it still 3? Let's see: 3
```

Exercise 5.1. If you are doing the prime project (chapter **??**) you can now do exercise **??**.

Background Square roots through Newton

Early computers had no hardware for computing a square root. Instead, they used *Newton's method*. Suppose you have a value y and you want want to compute $x \leftarrow \sqrt{y}$. This is equivalent to finding the zero of

$$f(x) = x^2 - y$$

where y is fixed. To indicate this dependence on y, we will write $f_y(x)$. Newton's method then finds the zero by evaluating

$$x_{\text{next}} = x - f_y(x)/f'_y(x)$$

until the guess is accurate enough, that is, until $f_y(x) \approx 0$.

Exercise 5.2.

- Write functions `f(x,y)` and `deriv(x,y)`, that compute $f_y(x)$ and $f'_y(x)$ for the definition of f_y above.
- Read a value y and iterate until $|\texttt{f}(\texttt{x},\texttt{y})| < 10^{-5}$. Print x.
- Second part: write a function `newton_root` that computes $\sqrt{y}$.

5.0.2 Pass by reference

Having only one output is a limitation on functions. Therefore there is a mechanism for altering the input parameters and returning (possibly multiple) results that way. You do this by not copying values into the function parameters, but by turning the function parameters into aliases of the variables at the place where the function is called.

We need the concept of a *reference*:

Reference

A reference is indicated with an ampersand in its definition, and it acts as an alias of the thing it references.

Code:

```
int i;
int &ri = i;
i = 5;
cout << i << "," << ri << endl;
i *= 2;
cout << i << "," << ri << endl;
ri -= 3;
cout << i << "," << ri << endl;
```

Output [basic] ref:

```
5,5
10,10
7,7
```

(You will not use references often this way.)

You can make a function parameter into a reference of a variable in the main program:

Parameter passing by reference

The function parameter `n` becomes a reference to the variable `i` in the main program:

```
void f(int &n) {
  n = /* some expression */ ;
};
int main() {
  int i;
  f(i);
  // i now has the value that was set in the function
}
```

Using the ampersand, the parameter is *passed by reference*: instead of copying the value, the function receives a *reference*: information where the variable is stored in memory.

Remark 2 *The* pass by reference mechanism in C *was different and should not be used in C++. In fact it was not a true pass by reference, but passing an address by value.*

We also the following terminology for function parameters:

- *input* parameters: passed by value, so that it only functions as input to the function, and no result is output through this parameter;
- *output* parameters: passed by reference so that they return an 'output' value to the program.
- *throughput* parameters: these are passed by reference, and they have an initial value when the function is called. In C++, unlike Fortran, there is no real separate syntax for these.

Pass by reference example 1

Code:

```
void f( int &i ) {
  i = 5;
}
int main() {

  int var = 0;
  f(var);
  cout << var << endl;
```

Output [basic] setbyref:

```
5
```

Compare the difference with leaving out the reference.

As an example, consider a function that tries to read a value from a file. With anything file-related, you always have to worry about the case of the file not existing and such. So our function returns:

- a boolean value to indicate whether the read succeeded, and
- the actual value if the read succeeded.

The following is a common idiom, where the success value is returned through the `return` statement, and the value through a parameter.

Pass by reference example 2

```
bool can_read_value( int &value ) {
  int file_status = try_open_file();
  if (file_status==0)
    value = read_value_from_file();
  return file_status!=0;
}

int main() {
  int n;
  if (!can_read_value(n))
    // if you can't read the value, set a default
    n = 10;
```

Exercise 5.3. Write a `void` function `swapij` of two parameters that exchanges the input values:

```
int i=2,j=3;
swapij(i,j);
// now i==3 and j==2
```

Exercise 5.4. Write a `bool` function that tests divisibility and returns a remainder:

```
int number,divisor,remainder;
// read in the number and divisor
if ( is_divisible(number,divisor,remainder) )
  cout << number << " is divisible by " << divisor << endl;
else
  cout << number << "/" << divisor <<
      " has remainder " << remainder << endl;
```

Exercise 5.5. If you are doing the geometry project, you should now do the exercises in section **??**.

Const parameters

You can prevent local changes to the function parameter:

```
/* This does not compile:
   void change_const_scalar(const int i) { i += 1; }
*/
```

This is mostly to protect you against yourself.

5.1 Sources used in this chapter

Listing of code/func/passvalue:

```
/****************************************************************
 ****
 **** This file belongs with the course
 **** Introduction to Scientific Programming in C++/Fortran2003
 **** copyright 2017/8 Victor Eijkhout eijkhout@tacc.utexas.edu
 ****
 **** passvalue.cxx : illustration pass-by-value
 ****
 ****************************************************************/

#include <iostream>
#include <cmath>
using std::cout;
using std::endl;

//examplesnippet passvalue
double squared( double x ) {
  x = x*x;
  return x;
}
//examplesnippet end

int main() {

  double number,other;
  //examplesnippet passvalue
  number = 5.1;
  cout << "Input starts as: "
       << number << endl;
  other = squared(number);
  cout << "Input var is now: "
       << number << endl;
  cout << "Output var is: "
       << other << endl;
  //examplesnippet end

  return 0;
}
```

Listing of code/func/localparm:

```
/****************************************************************
 ****
 **** This file belongs with the course
 **** Introduction to Scientific Programming in C++/Fortran2003
 **** copyright 2016-8 Victor Eijkhout eijkhout@tacc.utexas.edu
 ****
 **** localparm.cxx : simple parameter passing
 ****
 ****************************************************************/
```

```
#include <iostream>
using std::cout;
using std::endl;

#include <vector>
using std::vector;

//examplesnippet localparm
void change_scalar(int i) { i += 1; }
//examplesnippet end

int main() {

  int number;

  //examplesnippet localparm
  number = 3;
  cout << "Number is 3: " << number << endl;
  change_scalar(number);
  cout << "is it still 3? Let's see: " << number << endl;
  //examplesnippet end

  return 0;
}
```

Listing of code/basic/ref:

```
/****************************************************************
 ****
 **** This file belongs with the course
 **** Introduction to Scientific Programming in C++/Fortran2003
 **** copyright 2018 Victor Eijkhout eijkhout@tacc.utexas.edu
 ****
 **** ref.cxx : using references, not as parameter
 ****
 ****************************************************************/

#include <iostream>
using std::cin;
using std::cout;
using std::endl;

int main() {

  //codesnippet refint
  int i;
  int &ri = i;
  i = 5;
  cout << i << "," << ri << endl;
  i *= 2;
  cout << i << "," << ri << endl;
  ri -= 3;
  cout << i << "," << ri << endl;
  //codesnippet end

  return 0;
```

```
}
```

Listing of code/basic/setbyref:

```
/*****************************************************************
 ****
 **** This file belongs with the course
 **** Introduction to Scientific Programming in C++/Fortran2003
 **** copyright 2018 Victor Eijkhout eijkhout@tacc.utexas.edu
 ****
 **** setbyref.cxx : illustration of passing by ref
 ****
 *****************************************************************/

#include <iostream>
using std::cout;
using std::endl;

//codesnippet setbyref
void f( int &i ) {
  i = 5;
}
int main() {

  int var = 0;
  f(var);
  cout << var << endl;
  //codesnippet end

  return 0;
}
```

Chapter 6

References

6.1 Reference

This section contains further facts about references, which you have already seen as a mechanism for parameter passing; section 4.0.2. Make sure you study that material first.

Passing a variable to a routine passes the value; in the routine, the variable is local.

```
void change_scalar(int i) { i += 1; }
  /* ... */
  number = 3;
  cout << "Number is 3: " << number << endl;
  change_scalar(number);
  cout << "is it still 3? Let's see: " << number << endl;
```

If you do want to make the change visible in the *calling environment*, use a reference:

```
void change_scalar_by_reference(int &i) { i += 1; }
```

There is no change to the calling program. (Some people who are used to C find this bad, since you can not see from the use of a function whether it passes *by reference* or *by value*.)

6.2 Pass by reference

If you use a mathematical style of subprograms, where some values go in, and a new entity comes out, in effect all the inputs can be copied. This style is called *functional programming*, and there is much to be said for it. For instance, it makes it possible for the compiler to reason about your program. The only thing you have to worry about is the cost of copying, if the inputs are of non-trivial size, such as arrays.

However, sometimes you want to alter the inputs, so instead of a copy you need a way of accessing the actual input object. That's what *references* are invented for: to allow a subprogram access to the actual input entity.

A bonus of using references is that you do not incur the cost of copying. So what if you want this efficiency, but your program is really functional in design? Then you can use a *const reference*: the

argument is passed by reference, but you indicate explicitly that the subprogram does not alter it, again allowing compiler optimizations.

Reference: change argument

A reference makes the function parameter a synonym of the argument.

```
void f( int &i ) { i += 1; };
int main() {
  int i = 2;
  f(i); // makes it 3
```

Reference: save on copying

```
class BigDude {
public:
   vector<double> array(5000000);
}

void f(BigDude d) {
  cout << d.array[0];
};

int main() {
   BigDude big;
   f(big); // whole thing is copied
```

Instead write:

```
void f( BigDude &thing ) { .... };
```

Prevent changes:

```
void f( const BigDude &thing ) { .... }
```

6.3 Reference to class members

Here is the naive way of returning a class member:

```
class Object {
private:
  SomeType thing;
public:
  SomeType get_thing() {
    return thing; };
};
```

The problem here is that the return statement makes a copy of `thing`, which can be expensive. Instead, it is better to return the member by *reference*:

```
  SomeType &get_thing() {
    return thing; };
```

The problem with this solution is that the calling program can now alter the private member. To prevent that, use a *const reference*:

Code:

```
class has_int {
private:
  int mine{1};
public:
  const int& int_to_get() { return mine; };
  int& int_to_set() { return mine; };
  void inc() { mine++; };
};
  /* ... */
  has_int an_int;
  an_int.inc(); an_int.inc(); an_int.inc();
  cout << "Contained int is now: "
       << an_int.int_to_get() << endl;
  /* Compiler error: an_int.int_to_get() = 5; */
  an_int.int_to_set() = 17;
  cout << "Contained int is now: "
       << an_int.int_to_get() << endl;
```

Output [const] constref:

```
Contained int is now: 4
Contained int is now: 17
```

In the above example, the function giving a reference was used in the left-hand side of an assignment. If you would use it on the right-hand side, you would not get a reference. The result of an expression can not be a reference.

Let's again make a class where we can get a reference to the internals:

```
class myclass {
private:
  int stored{0};
public:
  myclass(int i) : stored(i) {};
  int &data() { return stored; };
};
```

Now we explore various ways of using that reference on the right-hand side:

Code:

```
myclass obj(5);
cout << "object data: "
     << obj.data() << endl;
int dcopy = obj.data();
dcopy++;
cout << "object data: "
     << obj.data() << endl;
int &dref = obj.data();
dref++;
cout << "object data: "
     << obj.data() << endl;
auto dauto = obj.data();
dauto++;
cout << "object data: "
     << obj.data() << endl;
auto &aref = obj.data();
aref++;
cout << "object data: "
     << obj.data() << endl;
```

Output [func] rhsref:

```
object data: 5
object data: 5
object data: 6
object data: 6
object data: 7
```

You see that, despite the fact that the method `data` was defined as returning a reference, you still need to indicate whether the left-hand side is a reference.

See section **??** for the interaction between `const` and references.

6.4 Reference to array members

You can define various operator, such as `+-*/` arithmetic operators, to act on classes, with your own provided implementation; see section 6.1.8. You can also define the parentheses and square brackets operators, so make your object look like a function or an array respectively.

These mechanisms can also be used to provide safe acess to arrays and/or vectors that are private to the object.

Suppose you have an object that contains an `int` array. You can return an element by defining the subscript (square bracket) operator for the class:

```
class vector10 {
private:
  int array[10];
public:
  /* ... */
  int operator()(int i) {
    return array[i];
  };
  int operator[](int i) {
    return array[i];
  };
};
  /* ... */
```

```
  vector10 v;
  cout << v(3) << endl;
  cout << v[2] << endl;
  /* compilation error: v(3) = -2; */
```

Note that `return array[i]` will return a copy of the array element, so it is not possible to write

```
myobject[5] = 6;
```

For this we need to return a reference to the array element:

```
  int& operator[](int i) {
    return array[i];
  };
};
  /* ... */
  cout << v[2] << endl;
  v[2] = -2;
  cout << v[2] << endl;
```

Another reason for wanting to return a reference is to prevent the *copy of the return result* that is induced by the `return` statement. In this case, you may not want to be able to alter the object contents, so you can return a *const reference*:

```
  const int& operator[](int i) {
    return array[i];
  };
};
  /* ... */
  cout << v[2] << endl;
  /* compilation error: v[2] = -2; */
```

6.5 rvalue references

See the chapter about obscure stuff; section **??**.

6.6 Sources used in this chapter

Listing of code/const/constref:

```
/****************************************************************
 ****
 **** This file belongs with the course
 **** Introduction to Scientific Programming in C++/Fortran2003
 **** copyright 2016-8 Victor Eijkhout eijkhout@tacc.utexas.edu
 ****
 **** constref.cxx : returning a const by ref
```

```
 ****
 ****************************************************************/

#include <iostream>
using std::cin;
using std::cout;
using std::endl;

//codesnippet constref
class has_int {
private:
  int mine{1};
public:
  const int& int_to_get() { return mine; };
  int& int_to_set() { return mine; };
  void inc() { mine++; };
};
//codesnippet end

int main() {

//codesnippet constref
  has_int an_int;
  an_int.inc(); an_int.inc(); an_int.inc();
  cout << "Contained int is now: "
       << an_int.int_to_get() << endl;
  /* Compiler error: an_int.int_to_get() = 5; */
  an_int.int_to_set() = 17;
  cout << "Contained int is now: "
       << an_int.int_to_get() << endl;
  //codesnippet end

  return 0;
}
```

Listing of code/func/rhsref:

```
/****************************************************************
 ****
 **** This file belongs with the course
 **** Introduction to Scientific Programming in C++/Fortran2003
 **** copyright 2018 Victor Eijkhout eijkhout@tacc.utexas.edu
 ****
 **** rhsref.cxx : result of an expression can not be reference
 ****
 ****************************************************************/

#include <iostream>
using std::cout;
using std::endl;

//codesnippet rhsrefclass
class myclass {
private:
  int stored{0};
public:
  myclass(int i) : stored(i) {};
```

```
  int &data() { return stored; };
};
//codesnippet end

int main() {

  //codesnippet rhsref
  myclass obj(5);
  cout << "object data: "
       << obj.data() << endl;
  int dcopy = obj.data();
  dcopy++;
  cout << "object data: "
       << obj.data() << endl;
  int &dref = obj.data();
  dref++;
  cout << "object data: "
       << obj.data() << endl;
  auto dauto = obj.data();
  dauto++;
  cout << "object data: "
       << obj.data() << endl;
  auto &aref = obj.data();
  aref++;
  cout << "object data: "
       << obj.data() << endl;
  //codesnippet end

  return 0;
}
```

Chapter 7

Classes and objects

7.1 What is an object?

You learned about `structs` (chapter **??**) as a way of abstracting from the elementary data types. The elements of a structure were called its members.

You also saw that it is possible to write functions that work on structures. Since these functions are really tied to the definition of the `struct`, shouldn't there be a way to make that tie explicitly?

That's what an object is:

- An object is like a structure in that it has data members.
- An object has *methods* which are the functions that operate on that object.

C++ does not actually have a 'object' keyword; instead you define a class with the `class` keyword, which describes all the objects of that class.

First of all, you can make an object look pretty much like a structure:

Code:

```
class Vector {
public:
  double x,y;
};

int main() {
  Vector p1;
  p1.x = 1.; p1.y = 2.; // This Is Not A Good Idea. See later.
  cout << "sum of components: " << p1.x+p1.y  << endl;
```

Output [geom] pointstruct:

```
sum of components: 3
```

Observations about the above code snippet:

- Again we have separate definition of the class and declaration of the objects. You define the class only once, after which you can make as many objects of that class as you want.
- There are data members. We will get to the `public` in a minute.
- You make an object of that class by using the class name as the datatype.
- The data members can be accessed with the period.

7.1.1 Constructor

Next we'll look at a syntax for creating class objects that is new. If you create an object, you actually call a function that has the same name as the class: the *constructor*. Above you had a declaration `Vector`

`p1` which was not just a declaration: it called the so-called *default constructor*, which has no arguments, and does nothing.

Usually you write your own constructor, for instance to initialize data members:

```
class Vector {
private:
  double vx,vy;
public:
  // constructor
  Vector( double userx,double usery ) {
    vx = userx; vy = usery;
  }

Vector p1(1.,2.), p2(3.7,-21./5);
```

7.1.2 Public and private

In the first example above, the data members of the `Vector` class were declared `public`, meaning that they are accessable from the calling (main) program. While this is initially convenient for coding, it is a bad idea in the long term. For a variety of reasons it is good practice to separate interface and implementation of a class.

Example of accessor functions

Getting and setting of members values is done through accessor functions:

```
class Vector {
private: // recommended!
  double vx,vy;
public:
  Vector( double x,double y ) {
    vx = x; vy = y;
  };

public:
  double x() { return vx; };
  double y() { return vy; };
  void setx( double newx ) {
    vx = newx; };
  void sety( double newy ) {
    vy = newy; };

}; // end of class definition

Vector p1(1.,2.);
```

Usage:

```
p1.setx(3.12);
/* ILLEGAL: p1.x() = 5; */
cout << "P1's x=" << p1.x() << endl;
```

Public versus private

- Implementation: data members, keep `private`,

- Interface: `public` functions to get/set data.
- Protect yourself against inadvertant changes of object data.
- Possible to change implementation without rewriting calling code.

You should not write access functions lightly: you should first think about what elements of your class should conceptually be inspectable or changeable by the outside world. Consider for example a class where a certain relation holds between members. In that case only changes are allowed that maintain that relation.

Private access gone wrong

We make a class with two members that sum to one.
You don't want to be able to change just one of them!

```
class SumIsOne {
public:
  float x,y;
  SumIsOne( double xx ) { x = xx; y = 1-x; };
}
int main() {
  SumIsOne pointfive(.5);
  pointfive.y = .6;
}
```

In general: enforce predicates on the members.

7.1.3 Initialization

Member default values

Class members can have default values, just like ordinary variables:

```
class Point {
private:
  float x=3., y=.14;
private:
  // et cetera
}
```

Each object will have its members initialized to these values.

Member initialization

Other syntax for initialization:

```
class Vector {
private:
  double x,y;
public:
  Vector( double userx,double usery ) : x(userx),y(usery) {
  }
```

Allows for reuse of names:

Code:

```
class Vector {
private:
  double x,y;
public:
  Vector( double x,double y ) : x(x),y(y) {
  }
  /* ... */
  Vector p1(1.,2.);
  cout << "p1 = "
       << p1.getx() << "," << p1.gety()
       << endl;
```

Output [geom] pointinitxy:

```
p1 = 1,2
```

Initializer lists can be used as denotations.

```
Point(float ux,float uy) {
/* ... */
Rectangle(Point bl,Point tr) {
/* ... */
Rectangle lielow( {0,0}, {5,2} );
```

7.1.4 Methods

With the accessors, you have just seen a first example of a class *method*: a function that is only defined for objects of that class, and that have access to the private data of that object. Let's now look at more meaningful methods. For instance, for the `Vector` class you can define functions such as `length` and `angle`.

Code:

```
class Vector {
private:
  double vx,vy;
public:
  Vector( double x,double y ) {
    vx = x; vy = y;
  };
  double length() { return sqrt(vx*vx + vy*vy); };
  double angle() { return 0.; /* something trig */; };
};

int main() {
  Vector p1(1.,2.);
  cout << "p1 has length " << p1.length() << endl;
```

Output [geom] pointfunc:

```
p1 has length 2.23607
```

Exercise 7.1. Add a `print` function to the `Vector` class.
How many parameters does it need?

By making these functions public, and the data members private, you define an Application Programmer Interface (API) for the class:

- You are defining operations for that class; they are the only way to access the data of the object.
- The methods can use the data of the object, or alter it. All data members, even when declared `private`, are global to the methods.

- Data members declared `private` are not accessible from outside the object.

So far you have seen methods that use the data members of an object to return some quantity. It is also possible to alter the members. For instance, you may want to scale a vector by some amount:

Code:

```
class Vector {
  /* ... */
  void scaleby( double a ) {
    vx *= a; vy *= a; };
  /* ... */
};
  /* ... */
  Vector p1(1.,2.);
  cout << "p1 has length " << p1.length() << endl;
  p1.scaleby(2.);
  cout << "p1 has length " << p1.length() << endl;
```

Output [geom] pointscaleby:

```
p1 has length 2.23607
p1 has length 4.47214
```

Direct alteration of internals

Return a reference to a private member:

```
class Vector {
private:
  double vx,vy;
public:
  double &x() { return vx; };
};
int main() {
  Vector v;
  v.x() = 3.1;
}
```

Reference to internals

Returning a reference saves you on copying.
Prevent unwanted changes by using a 'const reference'.

```
class Grid {
private:
  vector<Point> thepoints;
public:
  const vector<Point> &points() {
    return thepoints; };
};
int main() {
  Grid grid;
  cout << grid.points()[0];
  // grid.points()[0] = whatever ILLEGAL
}
```

The methods you have seen so far only returned elementary datatypes. It is also possible to return an object, even from the same class. For instance, instead of scaling the members of a vector object, you could create a new object based on the scaled members:

Code:

```
class Vector {
  /* ... */
  Vector scale( double a ) {
    return Vector( vx*a, vy*a ); };
  /* ... */
};
  /* ... */
  cout << "p1 has length " << p1.length() << endl;
  Vector p2 = p1.scale(2.);
  cout << "p2 has length " << p2.length() << endl;
```

Output [geom] pointscale:

```
p1 has length 2.23607
p2 has length 4.47214
```

7.1.5 Default constructor

One of the more powerful ideas in C++ is that there can be more than one constructor. You will often be faced with this whether you want or not. The following code looks plausible:

```
Vector p1(1.,2.), p2;
cout << "p1 has length " << p1.length() << endl;
p2 = p1.scale(2.);
cout << "p2 has length " << p2.length() << endl;
```

but it will give an error message during compilation. The reason is that

```
Vector p;
```

calls the default constructor. Now that you have defined your own constructor, the default constructor no longer exists. So you need to define it explicitly:

```
Vector() {};
Vector( double x,double y ) {
  vx = x; vy = y;
};
```

7.1.6 Accessors

It is a good idea to keep data members private, and use accessor functions.

Use accessor functions!

```
class PositiveNumber { /* ... */ }
class Point {
private:
  // data members
public:
  Point( float x,float y ) { /* ... */ };
```

```
  Point( PositiveNumber r,float theta ) { /* ... */ };
  float get_x() { /* ... */ };
  float get_y() { /* ... */ };
  float get_r() { /* ... */ };
  float get_theta() { /* ... */ };
};
```

Functionality is independent of implementation.

Exercise 7.2. If you are doing the prime project (chapter **??**), now is a good time to do exercise **??**.

The remainder of this section is advanced material. Make sure you have studied section 5.3.

It is a good idea to make the data in an object private, to control outside access to it.

- Sometimes this private data is auxiliary, and there is no reason for outside access.
- Sometimes you do want outside access, but you want to precisely control by whom.

Accessor functions:

```
class thing {
private:
 float x;
public:
 float get_x() { return x; };
 void set_x(float v) { x = v; }
};
```

This has advantages:

- You can print out any time you get/set the value; great for debugging:
  ```
  void set_x(float v) {
    cout << "setting: " << v << endl;
    x = v; }
  ```
- You can catch specific values: if `x` is always supposed to be positive, print an error (throw an exception) if nonpositive.

Having two accessors can be a little clumsy. Is it possible to use the same accessor for getting and setting?

Setting members through accessor

Use a single accessor for getting and setting:

Code:

```
class SomeObject {
private:
  float x=0.;
public:
  SomeObject( float v ) : x(v) {};
  float &xvalue() { return x; };
};

int main() {
  SomeObject myobject(1.);
  cout << "Object member initially :"
       << myobject.xvalue() << endl;
  myobject.xvalue() = 3.;
  cout << "Object member updated   :"
       << myobject.xvalue() << endl;
```

Output [object] accessref:

```
Object member initially :1
Object member updated   :3
```

The function `the_x` returns a reference to the internal variable `x`.

Of course you should only do this if you want the internal variable to be directly changable!

Exercise 7.3. This is a good time to do the exercises in section **??**.

7.1.7 Accessability

You can have multiple methods with the same name, as long as they can be distinguished by their argument types. This is known as *overloading*.

7.1.8 Operator overloading

Instead of writing

```
myobject.plus(anotherobject)
```

you can actually redefine the + operator so that

```
myobject + anotherobject
```

is legal.

The syntax for this is

Operator overloading

```
<returntype> operator<op>( <argument> ) { <definition> }
```

For instance:

```
class Point {
private:
  float x,y;
public:
  Point operator*(float factor) {
```

```
    return Point(factor*x,factor*y);
  };
};
```

Can even redefine equals and parentheses.

See section 5.4 for redefining the parentheses and square brackets.

Exercise 7.4. Write a `Fraction` class, and define the arithmetic operators on it.

7.1.9 Copy constructor

Just like the default constructor which is defined if you don't define an explicit constructor, there is an implicitly defined *copy constructor*. This constructor is invoked whenever you do an obvious copy:

```
my_object x,y; // regular or default constructor
x = y;         // copy constructor
```

Usually the copy constructor that is implicitly defined does the right thing: it copies all data members. (If you want to define your own copy constructor, you need to know its prototype. We will not go into that.)

Example of the copy constructor in action:

```
class has_int {
private:
  int mine{1};
public:
  has_int(int v) {
    cout << "set: " << v << endl;
    mine = v; };
  has_int( has_int &h ) {
    auto v = h.mine;
    cout << "copy: " << v << endl;
    mine = v; };
  void printme() { cout
      << "I have: " << mine << endl; };
};
```

Code:

```
has_int an_int(5);
has_int other_int(an_int);
an_int.printme();
other_int.printme();
```

Output [object] copyscalar:

```
set: 5
copy: 5
I have: 5
I have: 5
```

Copying is recursive

Class with a vector:

```
class has_vector {
private:
```

```
  vector<int> myvector;
public:
  has_vector(int v) { myvector.push_back(v); };
  void set(int v) { myvector.at(0) = v; };
  void printme() { cout
      << "I have: " << myvector.at(0) << endl; };
};
```

Copying is recursive, so the copy has its own vector:

Code:

```
has_vector a_vector(5);
has_vector other_vector(a_vector);
a_vector.set(3);
a_vector.printme();
other_vector.printme();
```

Output [object] copyvector:

```
I have: 3
I have: 5
```

7.1.10 Destructor

Just there is a constructor routine to create an object, there is a *destructor* to destroy the object. As with the case of the default constructor, there is a default destructor, which you can replace with your own.

A destructor can be useful if your object contains dynamically created data: you want to use the destructor to dispose of that dynamic data to prevent a *memory leak*. Another example is closing files for which the *file handle* is stored in the object.

The destructor is typically called without you noticing it. For instance, any statically created object is destroyed when the control flow leaves its scope.

Example:

Code:

```
class SomeObject {
public:
  SomeObject() { cout <<
    "calling the constructor"
    << endl; };
  ~SomeObject() { cout <<
    "calling the destructor"
    << endl; };
};
  /* ... */
  cout << "Before the nested scope" << endl;
  {
    SomeObject obj;
    cout << "Inside the nested scope" << endl;
  }
  cout << "After the nested scope" << endl;
```

Output [object] destructor:

```
Before the nested scope
calling the constructor
Inside the nested scope
calling the destructor
After the nested scope
```

Exercise 7.5. Write a class

```
class HasInt {
private:
  int mydata;
```

```
public:
  HasInt(int v) { /* initialize */ };
  ...
}
```

used as

```
{ HasInt v(5);
  v.set(6);
  v.set(-2);
}
```

which gives output

```
**** creating object with 5 ****
**** setting object to 6 ****
**** setting object to -2 ****
**** object destroyed after 2 updates ****
```

Destructors and exceptions

The destructor is called when you throw an exception:

Code:

```
class SomeObject {
public:
  SomeObject() { cout <<
    "calling the constructor"
    << endl; };
  ~SomeObject() { cout <<
    "calling the destructor"
    << endl; };
};
  /* ... */
  try {
    SomeObject obj;
    cout << "Inside the nested scope" << endl;
    throw(1);
  } catch (...) {
    cout << "Exception caught" << endl;
  }
```

Output [object] exceptobj:

```
calling the constructor
Inside the nested scope
calling the destructor
Exception caught
```

7.2 Inclusion relations between classes

The data members of an object can be of elementary datatypes, or they can be objects. For instance, if you write software to manage courses, each `Course` object will likely have a `Person` object, corresponding to the teacher.

```
class Person { /* ... */ }
class Course {
private:
  Person the_instructor;
```

```
  int year;
}
class Person {
  string name;
  ....
}
```

Designing objects with relations between them is a great mechanism for writing structured code, as it makes the objects in code behave like objects in the real world. The relation where one object contains another, is called a *has-a relation* between classes.

7.2.1 Accessors and other methods

Sometimes a class can behave as if it includes an object of another class, while not actually doing so. For instance, a line segment can be defined from two points

```
class Segment {
private:
  Point starting_point,ending_point;
}
  ...
int main() {
  Segment somesegment;
  Point somepoint = somesegment.get_the_end_point();
```

or from one point, and a distance and angle:

```
class Segment {
private:
  Point starting_point;
  float length,angle;
}
```

In both cases the code using the object is written as if the segment object contains two points. This illustrates how object-oriented programming can decouple the API of classes from their actual implementation.

Related to this decoupling, a class can also have two very different constructors.

```
class Segment {
private:
 // up to you how to implement!
public:
  Segment( Point start,float length,float angle )
    { .... }
  Segment( Point start,Point end ) { ... }
```

Depending on how you actually implement the class, the one constructor will simply store the defining data, and the other will do some conversion from the given data to the actually stored data.

This is another strength of object-oriented programming: you can change your mind about the implementation of a class without having to change the program that uses the class.

Exercise 7.6. If you are doing the geometry project, this is a good time to do the exercises in section **??**.

7.3 Inheritance

In addition to the has-a relation, there is the *is-a relation*, also called *inheritance*. Here one class is a special case of another class. Typically the object of the *derived class* (the special case) then also inherits the data and methods of the *base class* (the general case).

```
class General {
protected: // note!
 int g;
public:
 void general_method() {};
};

class Special : public General {
public:
  void special_method() { ... g ... };
};
```

How do you define a derived class?

- You need to indicate what the base class is:
  `class Special : public General { .... }`
- The base class needs to declare its data members as `protected`: this is similar to private, except that they are visible to derived classes
- The methods of the derived class can then refer to data of the base class;
- Any method defined for the base class is available as a method for a derived class object.

The derived class has its own constructor, with the same name as the class name, but when it is invoked, it also calls the constructor of the base class. This can be the default constructor, but often you want to call the base constructor explicitly, with parameters that are describing how the special case relates to the base case. Example:

```
class General {
public:
  General( double x,double y ) {};
};
class Special : public General {
public:
  Special( double x ) : General(x,x+1) {};
```

```
};
```

7.3.1 Methods of base and derived classes

Overriding methods

- A derived class can inherit a method from the base class.
- A derived class can define a method that the base class does not have.
- A derived class can *override* a base class method:

```
class Base {
public:
  virtual f() { ... };
};
class Deriv : public Base {
public:
  virtual f() override { ... };
};
```

Exercise 7.7. If you are doing the geometry project, you can now do the exercises in section **??**.

7.3.2 Virtual methods

Base vs derived methods

- Method defined in base class: can be used in any derived class.
- Method define in derived class: can only be used in that particular derived class.
- Method defined both in base and derived class, marked `override`: derived class method replaces (or extends) base class method.
- Virtual method: base class only declares that a routine has to exist, but does not give base implementation.
  A class is called *abstract class* if it has virtual methods; pure virtual if all methods are virtual.
  You can not make abstract objects.

Abstract classes

Special syntax for *abstract method*:

```
class Base {
public:
  virtual void f() = 0;
};
class Deriv {
public:
  virtual void f() { ... };
};
```

Example: using virtual class

```
class VirtualVector {
private:
public:
  virtual void setlinear(float) = 0;
  virtual float operator[](int) = 0;
};
```

Suppose `DenseVector` derives from `VirtualVector`:

```
DenseVector v(5);
v.setlinear(7.2);
cout << v[3] << endl;
```

Implementation

```
class DenseVector : VirtualVector {
private:
  vector<float> values;
public:
  DenseVector( int size ) {
    values = vector<float>(size,0);
  };
  void setlinear( float v ) {
    for (int i=0; i<values.size(); i++)
      values[i] = i*v;
  };
  float operator[](int i) {
    return values.at(i);
  };
};
```

7.3.3 Advanced topics in inheritance

More

- Multiple inheritance: an X is-a A, but also is-a B.
  This mechanism is somewhat dangerous.
- Virtual base class: you don't actually define a function in the base class, you only say 'any derived class has to define this function'.
- Friend classes:

```
class A;
class B {
  friend class A;
private:
  int i;
};
class  A {
public:
  void f(B b) { b.i; };
};
```

  A `friend` class can access private data and methods even if there is no inheritance relationship.

7.4 More topics about classes

'this'

Inside an object, a *pointer* to the object is available as `this`:

```
class Myclass {
private:
  int myint;
public:
  Myclass(int myint) {
    this->myint = myint;
  };
};
```

This is not often needed. Typical use case: you need to call a function inside a method that needs the object as argument)

```
class someclass;
void somefunction(const someclass &c) {
  /* ... */ }
class someclass {
// method:
void somemethod() {
  somefunction(*this);
};
```

7.4.1 Static members

Static class members

A static member acts like shared between all objects.

```
class MyClass {
private:
  static object_count;
public:
  MyClass() { object_count++; };
}
```

Initialization has to be done elsewhere:

```
MyClass::object_count = 0;
```

7.5 Review question

Review 7.1. Fill in the missing term

- The functionality of a class is determined by its...
- The state of an object is determined by its...

How many constructors do you need to specify in an object definition?

- Zero

- Zero or more
- One
- One or more

Exercise 7.8. Describe various ways to initialize the members of an object.

7.6 Sources used in this chapter

Listing of code/geom/pointstruct:

```
/****************************************************************
 ****
 **** This file belongs with the course
 **** Introduction to Scientific Programming in C++/Fortran2003
 **** copyright 2017 Victor Eijkhout eijkhout@tacc.utexas.edu
 ****
 **** pointstruct.cxx : make a Point class look just like a struct
 ****
 ****************************************************************/

#include <iostream>
using std::cin;
using std::cout;
using std::endl;

//codesnippet pointstruct
class Vector {
public:
  double x,y;
};

int main() {
  Vector p1;
  p1.x = 1.; p1.y = 2.; // This Is Not A Good Idea. See later.
  cout << "sum of components: " << p1.x+p1.y  << endl;
//codesnippet end

  return 0;
}
```

Listing of code/geom/pointinitxy:

```
/****************************************************************
 ****
 **** This file belongs with the course
 **** Introduction to Scientific Programming in C++/Fortran2003
 **** copyright 2017/8 Victor Eijkhout eijkhout@tacc.utexas.edu
 ****
 **** pointinit.cxx : about object initialization
 ****
 ****************************************************************/

#include <iostream>
using std::cin;
using std::cout;
using std::endl;
```

```
//codesnippet classpointinitxy
class Vector {
private:
  double x,y;
public:
  Vector( double x,double y ) : x(x),y(y) {
  }
//codesnippet end
  double getx() { return x; };
  double gety() { return y; };
};

int main() {
//codesnippet classpointinitxy
  Vector p1(1.,2.);
  cout << "p1 = "
       << p1.getx() << "," << p1.gety()
       << endl;
//codesnippet end

  return 0;
}
```

Listing of code/geom/pointfunc:

```
/*****************************************************************
 ****
 **** This file belongs with the course
 **** Introduction to Scientific Programming in C++/Fortran2003
 **** copyright 2017/8 Victor Eijkhout eijkhout@tacc.utexas.edu
 ****
 **** pointfun.cxx : class with method
 ****
 *****************************************************************/

#include <iostream>
using std::cin;
using std::cout;
using std::endl;

#include <cmath>
using std::sqrt;

//codesnippet pointfunc
class Vector {
private:
  double vx,vy;
public:
  Vector( double x,double y ) {
    vx = x; vy = y;
  };
  double length() { return sqrt(vx*vx + vy*vy); };
  double angle() { return 0.; /* something trig */; };
};

int main() {
```

```
  Vector p1(1.,2.);
  cout << "p1 has length " << p1.length() << endl;
//codesnippet end

  return 0;
}
```

Listing of code/geom/pointscaleby:

```
/*****************************************************************
 ****
 **** This file belongs with the course
 **** Introduction to Scientific Programming in C++/Fortran2003
 **** copyright 2017/8 Victor Eijkhout eijkhout@tacc.utexas.edu
 ****
 **** pointscaleby.cxx : method that operates on members
 ****
 *****************************************************************/

#include <iostream>
using std::cin;
using std::cout;
using std::endl;

#include <cmath>
using std::sqrt;

//codesnippet pointscaleby
class Vector {
//codesnippet end
private:
  double vx,vy;
public:
  Vector( double x,double y ) {
    vx = x; vy = y;
  };
//codesnippet pointscaleby
  void scaleby( double a ) {
    vx *= a; vy *= a; };
//codesnippet end
  double length() { return sqrt(vx*vx + vy*vy); };
//codesnippet pointscaleby
};
//codesnippet end

int main() {
//codesnippet pointscaleby
  Vector p1(1.,2.);
  cout << "p1 has length " << p1.length() << endl;
  p1.scaleby(2.);
  cout << "p1 has length " << p1.length() << endl;
//codesnippet end

  return 0;
}
```

Listing of code/geom/pointscale:

```
/****************************************************************
 ****
 **** This file belongs with the course
 **** Introduction to Scientific Programming in C++/Fortran2003
 **** copyright 2017 Victor Eijkhout eijkhout@tacc.utexas.edu
 ****
 **** pointscale.cxx : Vector class with private data
 ****
 ****************************************************************/

#include <iostream>
using std::cin;
using std::cout;
using std::endl;

#include <cmath>
using std::sqrt;

//codesnippet pointscale
class Vector {
//codesnippet end
private:
  double vx,vy;
public:
  Vector( double x,double y ) {
    vx = x; vy = y;
  };
//codesnippet pointscale
  Vector scale( double a ) {
    return Vector( vx*a, vy*a ); };
//codesnippet end
  double length() { return sqrt(vx*vx + vy*vy); };
//codesnippet pointscale
};
//codesnippet end

int main() {
  Vector p1(1.,2.);
//codesnippet pointscale
  cout << "p1 has length " << p1.length() << endl;
  Vector p2 = p1.scale(2.);
  cout << "p2 has length " << p2.length() << endl;
//codesnippet end

  return 0;
}
```

Listing of code/object/accessref:

```
/****************************************************************
 ****
 **** This file belongs with the course
 **** Introduction to Scientific Programming in C++/Fortran2003
 **** copyright 2016-8 Victor Eijkhout eijkhout@tacc.utexas.edu
 ****
 **** accessref.cxx : method returning reference
 ****
```

```
 ****************************************************************/

#include <iostream>
using std::cout;
using std::endl;

//codesnippet objaccessref
class SomeObject {
private:
  float x=0.;
public:
  SomeObject( float v ) : x(v) {};
  float &xvalue() { return x; };
};

int main() {
  SomeObject myobject(1.);
  cout << "Object member initially :"
       << myobject.xvalue() << endl;
  myobject.xvalue() = 3.;
  cout << "Object member updated   :"
       << myobject.xvalue() << endl;
  //codesnippet end

  return 0;
}
```

Listing of code/object/copyscalar:

```
/****************************************************************
 ****
 **** This file belongs with the course
 **** Introduction to Scientific Programming in C++/Fortran2003
 **** copyright 2017/8 Victor Eijkhout eijkhout@tacc.utexas.edu
 ****
 **** copyscalar.cxx : copy constructor with a simple class
 ****
 ****************************************************************/

#include <iostream>
using std::cin;
using std::cout;
using std::endl;

//codesnippet classwithcopy
class has_int {
private:
  int mine{1};
public:
  has_int(int v) {
    cout << "set: " << v << endl;
    mine = v; };
  has_int( has_int &h ) {
    auto v = h.mine;
    cout << "copy: " << v << endl;
    mine = v; };
  void printme() { cout
```

```
      << "I have: " << mine << endl; };
};
//codesnippet end

int main() {

  //codesnippet classwithcopyuse
  has_int an_int(5);
  has_int other_int(an_int);
  an_int.printme();
  other_int.printme();
  //codesnippet end

  return 0;
}
```

Listing of code/object/copyvector:

```
/****************************************************************
 ****
 **** This file belongs with the course
 **** Introduction to Scientific Programming in C++/Fortran2003
 **** copyright 2018 Victor Eijkhout eijkhout@tacc.utexas.edu
 ****
 **** copyvector.cxx : copy constructor with a class containing vector
 ****
 ****************************************************************/

#include <iostream>
using std::cin;
using std::cout;
using std::endl;
#include <vector>
using std::vector;

//codesnippet classwithcopyvector
class has_vector {
private:
  vector<int> myvector;
public:
  has_vector(int v) { myvector.push_back(v); };
  void set(int v) { myvector.at(0) = v; };
  void printme() { cout
      << "I have: " << myvector.at(0) << endl; };
};
//codesnippet end

int main() {

  //codesnippet classwithcopyvectoruse
  has_vector a_vector(5);
  has_vector other_vector(a_vector);
  a_vector.set(3);
  a_vector.printme();
  other_vector.printme();
  //codesnippet end
```

```
  return 0;
}
```

Listing of code/object/destructor:

```
/*****************************************************************
 ****
 **** This file belongs with the course
 **** Introduction to Scientific Programming in C++/Fortran2003
 **** copyright 2016-8 Victor Eijkhout eijkhout@tacc.utexas.edu
 ****
 **** destructor.cxx : illustration of objects going out of scope
 ****
 *****************************************************************/

#include <iostream>
using std::cin;
using std::cout;
using std::endl;

//examplesnippet destructor
class SomeObject {
public:
  SomeObject() { cout <<
    "calling the constructor"
    << endl; };
  ~SomeObject() { cout <<
    "calling the destructor"
    << endl; };
};
//examplesnippet end

int main() {

//examplesnippet destructor
  cout << "Before the nested scope" << endl;
  {
    SomeObject obj;
    cout << "Inside the nested scope" << endl;
  }
  cout << "After the nested scope" << endl;
//examplesnippet end

  return 0;
}
```

Listing of code/object/exceptobj:

Chapter 8

Pointers

The term pointer is used to denote a reference to a quantity. This chapter will explain pointers, and give some uses for them. If you don't already speak C, skip to the first subsection.

If you do already know C, we remark that there is less need for pointers in C++ than there was in C.

- To pass an argument *by reference*, use a *reference*. Section **??**.
- Strings are done through `std::string`, not character arrays; see above.
- Arrays can largely be done through `std::vector`, rather than `malloc`; see above.
- Traversing arrays and vectors can be done with ranges; section 2.1.2.
- Anything that obeys a scope should be created through a *constructor*, rather than using `malloc`.

Legitimate needs:

- Linked lists and Directed Acyclic Graphs (DAGs); see the example in section **??**.
- Objects on the heap.
- Use `nullptr` as a signal.

8.1 The 'arrow' notation

Members from pointer

- If `x` is object with member `y`:
  `x.y`
- If `xx` is pointer to object with member `y`:
  `xx->y`
- In class methods `this` is a pointer to the object, so:

```
class x {
  int y;
  x(int v) {
    this->y = v; }
}
```

- Arrow notation works with old-style pointers and new shared/unique pointers.

8.2 Making a shared pointer

Smart pointers are used the same way as old-style pointers in C. If you have an object `Obj X` with a member `y`, you access that with `X.y`; if you have a pointer `X` to such an object, you write `X->y`.

So what is the type of this latter `X` and how did you create it?

Creating a shared pointer

Allocation and pointer in one:

```
shared_ptr<Obj> X =
    make_shared<Obj>( /* constructor args */ );
  // or:
auto X = make_shared<Obj>( /* args */ );
  // or:
auto X = shared_ptr<Obj>( new Obj(/* args */) );

X->method_or_member;
```

This requires at the top of your file:

```
#include <memory>
using std::shared_ptr;
using std::make_shared;
```

Simple example

Code:

```
class HasX {
private:
  double x;
public:
  HasX( double x) : x(x) {};
  auto &val() { return x; };
};

int main() {
  auto X = make_shared<HasX>(5);
  cout << X->val() << endl;
  X->val() = 6;
  cout << X->val() << endl;
```

Output [pointer] pointx:

```
5
6
```

8.2.1 Pointers and arrays

Exercise 8.1. If you are doing the prime numbers project (chapter **??**) you can now do exercise **??**.

Exercise 8.2. The prototypical example use of pointers is in linked lists. You can now do the exercises in section **??**.

8.2.2 Smart pointers versus address pointers

Pointers don't go with addresses

The oldstyle `&y` address pointer can not be made smart:

```
auto
  p1 = shared_ptr<HasY>( &y ),
  p2 = shared_ptr<HasY>( &y );
p1->y = 3;
cout << "Pointer 2's y: "
     << p2->y << endl;
```

gives:

```
address(56325,0x7fff977cc380) malloc: *** error for object
    0x7ffeeb9caf08: pointer being freed was not allocated
```

8.3 Garbage collection

The big problem with C-style pointers is the chance of a *memory leak*. If a pointer to a block of memory goes out of scope, the block is not returned to the Operating System (OS), but it is no longer accessible.

```
// the variable 'array' doesn't exist
{
  // attach memory to 'array':
  double *array = new double[25];
  // do something with array
}
// the variable 'array' does not exist anymore
// but the memory is still reserved.
```

Shared and unique pointers do not have this problem: if they go out of scope, or are overwritten, the destructor on the object is called, thereby releasing any allocated memory.

An example.

Reference counting illustrated

We need a class with constructor and destructor tracing:

```
class thing {
public:
  thing()  { cout << ".. calling constructor\n"; };
  ~thing() { cout << ".. calling destructor\n"; };
};
```

Pointer overwrite

Let's create a pointer and overwrite it:

Code:

```
cout << "set pointer1"
     << endl;
auto thing_ptr1 =
  make_shared<thing>();
cout << "overwrite pointer"
     << endl;
thing_ptr1 = nullptr;
```

Output [pointer] ptr1:

```
set pointer1
.. calling constructor
overwrite pointer
.. calling destructor
```

However, if a pointer is copied, there are two pointers to the same block of memory, and only when both disappear, or point elsewhere, is the object deallocated.

Pointer copy

Code:

```
cout << "set pointer2" << endl;
auto thing_ptr2 =
  make_shared<thing>();
cout << "set pointer3 by copy"
     << endl;
auto thing_ptr3 = thing_ptr2;
cout << "overwrite pointer2"
     << endl;
thing_ptr2 = nullptr;
cout << "overwrite pointer3"
     << endl;
thing_ptr3 = nullptr;
```

Output [pointer] ptr2:

```
set pointer2
.. calling constructor
set pointer3 by copy
overwrite pointer2
overwrite pointer3
.. calling destructor
```

8.4 More about pointers

8.4.1 Get the pointed data

Most of the time, accessing the target of the pointer through the arrow notation is enough. However, if you actually want the object, you can get it with `get`. Note that this does not give you the pointed object, but a traditional pointer.

Getting the underlying pointer

```
X->y;
// is the same as
X.get()->y;
// is the same as
( *X.get() ).y;
```

Code:

```
auto Y = make_shared<HasY>(5);
cout << Y->y << endl;
Y.get()->y = 6;
cout << ( *Y.get() ).y << endl;
```

Output [pointer] pointy:

```
5
6
```

8.4.2 Pointers to non-objects

In the introduction to this chapter we argued that many of the uses for pointers that existed in C have gone away in C++, and the main one left is the case where multiple objects share 'ownership' of some other object.

You can still make shared pointers to scalar data, for instance to an array of scalars. You then get the advantage of the memory management, but you do not get the `size` function and such that you would have if you'd used a `vector` object.

Code:

```
auto array = make_shared<double>(50);
shared_ptr<double> other;
array.get()[2] = 3.;
// the following two are ILLEGAL:
// array->at(2) = 4.;
// array.get().at(2) = 4.;
other = array;
cout << other.get()[2] << endl;
```

Output [pointer] ptrarray:

```
3
```

8.4.3 Shared pointer to 'this'

Inside an object method, the object is accessible as `this`. This is a pointer in the classical sense. So what if you want to refer to 'this' but you need a shared pointer?

For instance, suppose you're writing a linked list code, and your `node` class has a method `prepend_or_append` that gives a shared pointer to the new head of the list.

Your code would start something like this, handling the case where the new node is appended to the current:

```
shared_pointer<node> node:append
    ( shared_ptr<node> other ) {
  if (other->value>this->value) {
    this->tail = other;
```

But now you need to return this node, as a shared pointer. But `this` is a `node*`, not a `shared_ptr<node>`.

The solution here is that you can return

```
return this->shared_from_this();
```

if you have defined your node class to inherit from what probably looks like magic:

```
class node : public enable_shared_from_this<node>
```

8.4.4 Null pointer

In C there was a macro *NULL* that, only by convention, was used to indicate *null pointers*: pointers that do not point to anything. C++ has the `nullptr`, which is an object of type `std::nullptr_t`.

There are some scenarios where this is useful, for instance, with polymorphic functions:

```
void f(int);
void f(int*);
```

Calling `f(ptr)` where the point is `NULL`, the first function is called, whereas with `nullptr` the second is called.

8.4.5 Example: linked lists

The standard example of pointer manipulation is 'linked lists'. This is discussed in some detail in section **??**.

8.5 Sources used in this chapter

Listing of code/pointer/pointx:

```
/****************************************************************
 ****
 **** This file belongs with the course
 **** Introduction to Scientific Programming in C++/Fortran2003
 **** copyright 2017/8 Victor Eijkhout eijkhout@tacc.utexas.edu
 ****
 **** pointx.cxx : access through arrow
 ****
 ****************************************************************/

#include <iostream>
using std::cout;
using std::endl;

#include <memory>
using std::make_shared;

//codesnippet pointx
class HasX {
private:
  double x;
public:
  HasX( double x) : x(x) {};
  auto &val() { return x; };
};

int main() {
  auto X = make_shared<HasX>(5);
  cout << X->val() << endl;
  X->val() = 6;
  cout << X->val() << endl;
  //codesnippet end
}
```

Listing of code/pointer/ptr1:

```
/****************************************************************
 ****
 **** This file belongs with the course
```

```
 **** Introduction to Scientific Programming in C++/Fortran2003
 **** copyright 2017/8 Victor Eijkhout eijkhout@tacc.utexas.edu
 ****
 **** ptr1.cxx : shared pointers
 ****
 ****************************************************************/

#include <iostream>
using std::cout;
using std::endl;

#include <memory>
using std::shared_ptr;
using std::make_shared;

//codesnippet thingcall
class thing {
public:
  thing()  { cout << ".. calling constructor\n"; };
  ~thing() { cout << ".. calling destructor\n"; };
};
//codesnippet end

int main() {

  //codesnippet shareptr1
  cout << "set pointer1"
       << endl;
  auto thing_ptr1 =
    make_shared<thing>();
  cout << "overwrite pointer"
       << endl;
  thing_ptr1 = nullptr;
  //codesnippet end

#if 0
  // alternatively
  auto thing_ptr1 = shared_ptr<thing>( new thing );
#endif

  return 0;
}
```

Listing of code/pointer/ptr2:

```
/****************************************************************
 ****
 **** This file belongs with the course
 **** Introduction to Scientific Programming in C++/Fortran2003
 **** copyright 2017/8 Victor Eijkhout eijkhout@tacc.utexas.edu
 ****
 **** ptr2.cxx : shared pointers
 ****
 ****************************************************************/

#include <iostream>
using std::cout;
```

```
using std::endl;

#include <memory>
using std::shared_ptr;
using std::make_shared;

class thing {
public:
  thing()  { cout << ".. calling constructor\n"; };
  ~thing() { cout << ".. calling destructor\n"; };
};

int main() {

  //codesnippet shareptr2
  cout << "set pointer2" << endl;
  auto thing_ptr2 =
    make_shared<thing>();
  cout << "set pointer3 by copy"
       << endl;
  auto thing_ptr3 = thing_ptr2;
  cout << "overwrite pointer2"
       << endl;
  thing_ptr2 = nullptr;
  cout << "overwrite pointer3"
       << endl;
  thing_ptr3 = nullptr;
  //codesnippet end

#if 0
  // alternatively
  auto thing_ptr2 =
    shared_ptr<thing>
      ( new thing );
#endif

  return 0;
}
```

Listing of code/pointer/pointy:

```
/****************************************************************
 ****
 **** This file belongs with the course
 **** Introduction to Scientific Programming in C++/Fortran2003
 **** copyright 2017/8 Victor Eijkhout eijkhout@tacc.utexas.edu
 ****
 **** pointx.cxx : access through arrow
 ****
 ****************************************************************/

#include <iostream>
using std::cout;
using std::endl;

#include <memory>
using std::make_shared;
```

```
class HasY {
public:
  double y;
  HasY( double y) : y(y) {};
};

int main() {
  //codesnippet pointy
  auto Y = make_shared<HasY>(5);
  cout << Y->y << endl;
  Y.get()->y = 6;
  cout << ( *Y.get() ).y << endl;
  //codesnippet end
}
```

Listing of code/pointer/ptrarray:

```
/****************************************************************
 ****
 **** This file belongs with the course
 **** Introduction to Scientific Programming in C++/Fortran2003
 **** copyright 2017/8 Victor Eijkhout eijkhout@tacc.utexas.edu
 ****
 **** ptrarray.cxx : pointer to C style array
 ****
 ****************************************************************/

#include <iostream>
using std::cout;
using std::endl;

#include <memory>
using std::shared_ptr;
using std::make_shared;

int main() {

  //codesnippet ptrarray
  auto array = make_shared<double>(50);
  shared_ptr<double> other;
  array.get()[2] = 3.;
  // the following two are ILLEGAL:
  // array->at(2) = 4.;
  // array.get().at(2) = 4.;
  other = array;
  cout << other.get()[2] << endl;
  //codesnippet end

  return 0;
}
```

Chapter 9

Namespaces

9.1 Solving name conflicts

In section 2.1.3 you saw that the C++ library comes with a *vector* class, that implements dynamic arrays. You say

```
std::vector<int> bunch_of_ints;
```

and you have an object that can store a bunch of ints. And if you use such vectors often, you can save yourself some typing by having

```
using namespace std;
```

somewhere high up in your file, and write

```
vector<int> bunch_of_ints;
```

in the rest of the file.

More safe:

```
using std::vector;
```

But what if you are writing a geometry package, which includes a vector class? Is there confusion with the Standard Template Library (STL) vector class? There would be if it weren't for the phenomenon *namespace*, which acts as as disambiguating prefix for classes, functions, variables.

You have already seen namespaces in action when you wrote `std::vector`: the '`std`' is the name of the namespace.

Defining a namespace

You can make your own namespace by writing

```
namespace a_namespace {
  // definitions
  class an_object {
  };
|
```

so that you can write

Namespace usage

```
a_namespace::an_object myobject();
```

or

```
using namespace a_namespace;
an_object myobject();
```

or

```
using a_namespace::an_object;
an_object myobject();
```

9.1.1 Namespace header files

If your namespace is going to be used in more than one program, you want to have it in a separate source file, with an accompanying header file:

```
#include "geolib.h"
using namespace geometry;
```

The header would contain the normal function and class headers, but now inside a named namespace:

```
namespace geometry {
  class point {
  private:
    double xcoord,ycoord;
  public:
    point() {};
    point( double x,double y );
    double x();
    double y();
  };
  class vector {
  private:
    point from,to;
  public:
    vector( point from,point to);
    double size();
  };
}
```

and the implementation file would have the implementations, in a namespace of the same name:

```
namespace geometry {
  point::point( double x,double y ) {
```

```
    xcoord = x; ycoord = y; };
  double point::x() { return xcoord; }; // 'accessor'
  double point::y() { return ycoord; };
  vector::vector( point from,point to) {
    this->from = from; this->to = to;
  };
  double vector::size() {
    double
      dx = to.x()-from.x(), dy = to.y()-from.y();
    return sqrt( dx*dx + dy*dy );
  };
}
```

9.2 Best practices

The problem with

```
using namespace std;
```

is that it may pull in unwanted functions. For instance:

Why not 'using namespace std'?

This compiles, but should not:

```
#include <iostream>
using namespace std;

int main() {
  int i=1,j=2;
  swap(i,j);
  cout << i << endl;
  return 0;
}
```

This gives an error:

```
#include <iostream>
using std::cout;
using std::endl;

int main() {
  int i=1,j=2;
  swap(i,j);
  cout << i << endl;
  return 0;
}
```

It is a good idea to pull in functions explicitly:

```
#include <iostream>
using std::cout;
```

In particular, one should never use the indiscriminate

```
using namespace std;
```

in a header file. Anyone using the header would have no idea what functions are suddenly defined.

Chapter 10

Templates

Sometimes you want a function or a class based on more than one different datatypes. For instance, in chapter 2 you saw how you could create an array of ints as `vector<int>` and of doubles as `vector<double>`. Here you will learn the mechanism for that.

Templated type name

Basically, you want the type name to be a variable. Syntax:

```
template <typename yourtypevariable>
// ... stuff with yourtypevariable ...
```

Historically `typename` was `class` but that's confusing.

10.1 Templated functions

Example: function

Definition:

```
template<typename T>
void function(T var) { cout << var << end; }
```

Usage:

```
int i; function(i);
double x; function(x);
```

and the code will behave as if you had defined `function` twice, once for `int` and once for `double`.

Exercise 10.1. Machine precision, or 'machine epsilon', is sometimes defined as the smallest number ϵ so that $1 + \epsilon > 1$ in computer arithmetic.
Write a templated function `epsilon` so that the following code prints out the values of the machine precision for the `float` and `double` type respectively:

Code:

```
float float_eps;
epsilon(float_eps);
cout << "For float, epsilon is "
     << float_eps << endl;

double double_eps;
epsilon(double_eps);
cout << "For double, epsilon is "
     << double_eps << endl;
```

Output [template] eps:

```
For float, epsilon is 9.999999e-08
For double, epsilon is 1.000000e-15
```

10.2 Templated classes

The most common use of templates is probably to define templated classes. You have in fact seen this mechanism in action:

Templated vector

the STL contains in effect

```
template<typename T>
class vector {
private:
  // data definitions omitted
public:
  T at(int i) { /* return element i */ };
  int size() { /* return size of data */ };
  // much more
}
```

10.3 Specific implementation

```
template <typename T>
void f(T);
template <>
void f(char c) { /* code with c */ };
template <>
void f(double d) { /* code with d */ };
```

10.4 Templating over non-types

THESE EXAMPLES ARE NOT GOOD.

See: https://www.codeproject.com/Articles/257589/An-Idiots-Guide-to-Cplusplus-Temp

Templating a value

Templeting over integral types, not double.

The templated quantity is a value:

```
template<int s>
std::vector<int> svector(s);
  /* ... */
  svector(3) threevector;
  cout << threevector.size();
```

Exercise 10.2. Write a class that contains an array. The length of the array should be templated.

10.5 Sources used in this chapter

Listing of code/template/eps:

```
/****************************************************************
 ****
 **** This file belongs with the course
 **** Introduction to Scientific Programming in C++/Fortran2003
 **** copyright 2016-8 Victor Eijkhout eijkhout@tacc.utexas.edu
 ****
 **** eps.cxx : a templated program for machine epsilon
 ****
 ****************************************************************/

#include <iostream>
using std::cin;
using std::cout;
using std::endl;
#include <iomanip>
using std::scientific;

//answersnippet epsprinter
template<typename realtype>
void epsilon(realtype &eps) {
  realtype one = (realtype)1;
  realtype trial = one;
  while (one+trial>one) {
    eps = trial;
    trial /= (realtype)10;
  }
}
//answersnippet end

int main() {

  cout << scientific;
  //codesnippet epsprint
  float float_eps;
  epsilon(float_eps);
```

```
  cout << "For float, epsilon is "
       << float_eps << endl;

  double double_eps;
  epsilon(double_eps);
  cout << "For double, epsilon is "
       << double_eps << endl;
  //codesnippet end

  return 0;
}
```

Chapter 11

Closures

The C++11 mechanism of *lambda expression*s makes dynamic definition of functions possible.

Lambda expressions

```
[capture] ( inputs ) -> outtype { definition };
```

Example:

```
[] (float x,float y) -> float {
  return x+y; } ( 1.5, 2.3 )
```

Store lambda in a variable:

```
auto summing =
  [] (float x,float y) -> float {
  return x+y; };
cout << summing ( 1.5, 2.3 ) << endl;
```

A non-trivial use of lambdas uses the *capture* to fix one argument of a function. Let's say we want a function that computes exponentials for some fixed exponent value. We take the `cmath` function

```
pow( x,exponent );
```

and fix the exponent:

```
auto powerfunction = [exponent] (float x) -> float {
  return pow(x,exponent); };
```

Now `powerfunction` is a function of one argument, which computes that argument to a fixed power.

Storing a lambda in a class is hard because it has unique type. Solution: use `std::function`

Lambda in object

```
#include <functional>
using std::function;
  /* ... */
class SelectedInts {
```

```
private:
  vector<int> bag;
  function< bool(int) > selector;
public:
  SelectedInts( function< bool(int) > f ) {
    selector = f; };
  void add(int i) {
    if (selector(i))
      bag.push_back(i);
  };
  int size() { return bag.size(); };
};
```

Illustration

```
SelectedInts greaterthan
  ( [threshold] (int i) -> bool { return i>threshold; } );
for (int i=0; i<upperbound; i++)
  greaterthan.add(i);
cout << "Ints under " << upperbound
     << " greater than " << threshold << ": "
     << greaterthan.size() << endl;
```

Exercise 11.1. Refer to 4.3 for background, and note that finding x such that $f(x) = a$ is equivalent to applying Newton to $f(x) - a$.
Implement a class `valuefinder` and its `double find(double)` method.

```
class valuefinder {
private:
  function< double(double) >
      f,fprime;
  double tolerance{.00001};
public:
  valuefinder
  ( function< double(double) > f,
    function< double(double) > fprime )
    : f(f),fprime(fprime) {};
```

used as

```
double root = newton_root.find(number);
```

Exercise 11.2. Can you write a derived class `rootfinder` used as

```
squarefinder newton_root;
double root = newton_root.find(number);
```

Chapter 12

Error handling

12.1 General discussion

When you're programming, making errors is close to inevitable. *Syntax errors*, violations of the grammar of the language, will be caught by the compiler, and prevent generation of an executable. In this section we will therefore talk about *runtime errors*: behaviour at runtime that is other than intended.

Here are some sources of runtime errors

Array indexing Using an index outside the array bounds may give a runtime error:

```
vector<float> a(10);
for (int i=0; i<=10; i++)
  a.at(i) = x; // runtime error
```

or undefined behaviour:

```
vector<float> a(10);
for (int i=0; i<=10; i++)
  a[i] = x;
```

See further section 2.1.3.

Null pointers Using an uninitialized pointer is likely to crash your program:

```
Object *x;
if (false) x = new Object;
x->method();
```

Numerical errors such as division by zero will not crash your program, so catching them takes some care.

Guarding against errors.

- Check preconditions.
- Catch results.
- Check postconditions.

Error reporting:

- Message
- Total abort
- Exception

Assertions:

```
#include <cassert>
...
assert( bool )
```

assertions are omitted with optimization

Function return values

12.2 Exception handling

Exception throwing

Throwing an *exception* is one way of signalling an error or unexpected behaviour:

```
void do_something() {
  if ( oops )
    throw(5);
}
```

Catching an exception

It now becomes possible to detect this unexpected behaviour by *catching* the exception:

```
try {
  do_something();
} catch (int i) {
  cout << "doing something failed: error=" << i << endl;
}
```

You can throw integers to indicate an error code, a string with an actual error message. You could even make an error class:

Exception classes

```
class MyError {
public :
  int error_no; string error_msg;
  MyError( int i,string msg )
  : error_no(i),error_msg(msg) {};
}

throw( MyError(27,"oops");

try {
  // something
} catch ( MyError &m ) {
  cout << "My error with code=" << m.error_no
    << " msg=" << m.error_msg << endl;
```

```
}
```

You can use exception inheritance!

Multiple catches

You can multiple `catch` statements to catch different types of errors:

```
try {
  // something
} catch ( int i ) {
  // handle int exception
} catch ( std::string c ) {
  // handle string exception
}
```

Catch any exception

Catch exceptions without specifying the type:

```
try {
  // something
} catch ( ... ) { // literally: three dots
  cout << "Something went wrong!" << endl;
}
```

Exercise 12.1. Define the function

$$f(x) = x^3 - 19x^2 + 79x + 100$$

and evaluate $\sqrt{f(i)}$ for the integers $i = 0 \ldots 20$.

- First write the program naively, and print out the root. Where is $f(i)$ negative? What does your program print?
- You see that floating point errors such as the root of a negative number do not make your program crash or something like that. Alter your program to throw an exception if $f(i)$ is negative, catch the exception, and print an error message.
- Alter your program to test the output of the `sqrt` call, rather than its input. Use the function `isnan`

  ```
  #include <cfenv>
  using std::isnan;
  ```

  and again throw an exception.

Exceptions in constructors

A *function try block* will catch exceptions, including in initializer lists of constructors.

```
f::f( int i )
  try : fbase(i) {
    // constructor body
```

```
}
catch (...) { // handle exception
}
```

More about exceptions

- Functions can define what exceptions they throw:
  ```
  void func() throw( MyError, std::string );
  void funk() throw();
  ```
- Predefined exceptions: `bad_alloc`, `bad_exception`, etc.
- An exception handler can throw an exception; to rethrow the same exception use '`throw;`' without arguments.
- Exceptions delete all stack data, but not `new` data. Also, destructors are called; section 6.1.10.
- There is an implicit `try/except` block around your main. You can replace the handler for that. See the `exception` header file.
- Keyword `noexcept`:
  ```
  void f() noexcept { ... };
  ```

12.3 Legacy mechanisms

The traditional approach to error checking is for each routine to return an integer parameter that indicates success or absence thereof. Problems with this approach arise if it's used inconsistently, for instance by a user forgetting to heed the return codes of a library. Also, it requires that every level of the function calling hierarchy needs to check return codes.

The *PETSc* library uses this mechanism consistently throughout, and to great effect.

Exceptions are a better mechanism, since

- they can not be ignored, and
- they do not require handling on the levels of the calling hierarchy between where the exception is thrown and where it is caught.

And then there is the fact that memory management is automatic with exceptions.

12.3.1 Legacy C mechanisms

The `errno` variable and the `setjmp` / `longjmp` functions should not be used. These functions for instance do not the memory management advantages of exceptions.

12.4 Tools

Despite all your careful programming, your code may still compute the wrong result or crash with strange errors. There are two tools that may then be of assistance:

- `gdb` is the GNU interactive *debugger*. With it, you can run your code step-by-step, inspecting variables along way, and detecting various conditions. It also allows you to inspect variables after your code throws an error.
- `valgrind` is a memory testing tool. It can detect memory leaks, as well as the use of uninitialized data.

Chapter 13

Index