Scientific Computing in C++23

Victor Eijkhout

2017–2022, formatted July 19, 2023
Book and slides download: https://tinyurl.com/vle322course
Public repository: https://bitbucket.org/VictorEijkhout/intro-programming-public

Contents

Chapter 1

Introduction

1.1 Reader profile, prerequisites.

This book does not start from zero: the following elementary knowledge of C++ is assumed.

We assume basic knowledge of

1. 1. C++ datatypes,
2. 2. control structures,
3. 3. functions
4. 4. object-oriented programming
5. 5. containers

We assume basic skills in:

1. 1. Designing and writing a program
2. 2. compiling
3. 3. running and debugging

1.2 Target knowledge level

This book teaches C++ techniques aimed at scientific computing. Corresponding to the above points, we cover:

1.2.1 Elements

Main program, commandline options, return code

1.2.2 C++ datatypes

Integer datatypes: problems with integer promotion of short types use of long integers for big data

Flaoat datatypes: basics of floating point arithmetic single vs double, precision

Complex

A word about vector-of-bool

1.2.3 Control structures

Initializer statement Range based loops Use of range header

1.2.4 Functions

Lambda expressions.

Spaceship operator

1.2.5 OO programming

Very modest discussion of templates expression templates, relating to point 5.?

Modules

1.2.6 Containers

Discussion of c++20 span and c++23 mdspan, use in linear algebra codes

1.2.7 misc

const, constexpr, constinit auto, decltype

1.2.8 Skills

1. Test-Driven Development and Unit Testing using Catch2
2. Use of build systems: Make and CMake
3. Use of debugger
4. Performance evaluation;
5. discussion of hardware issues and how they manifest in code;
6. programming techniques for overcoming hardware problems.
7. Misc: Use of libraries: cxxopts, Eigen (see containers above), GMP, GSL

1.3 Advanced topics

Parallelism through OpenMP

PART I

C++

Chapter 2

Basic elements of C++

2.1 The main program

The *main* program has to be of type **int**; however, many compilers tolerate deviations from this, for instance accepting **void**, which is not language standard.

The arguments to main can be:

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

The *argc/argv* variables contain the commandline as a set of strings.

- *argc* is the number of strings: the name of the program, and the number of space-separated arguments;
- *argv* contains the *commandline arguments* as an array of strings.

You might be tempted to parse the commandline ad-hoc, but there are dedicated libraries for this; see section 35.2.2.

The returned **int** can be specified several ways:

- If no **return** statement is given, implicitly **return** 0 is performed. (This is also true in C99.)
- You can explicitly pass an integer to the operating system, which can then be queried in the *shell* as a *return code*:

Code:

```
int main() {
  return 1;
}
```

Output
[basic] return:

```
./return ; \
        if [ $? -ne 0 ] ;
    then \
            echo "Program
    failed" ; \
        fi
Program failed
```

- For cleanliness, you can use the values *EXIT_SUCCESS* and *EXIT_FAILURE* which are defined in *cstdlib.h*.
- You can also use the *exit* function:

```
void exit(int);
```

2.2 Integers

2.2.1 Integers

There are several integer types, differing by the number of bytes they take up, and by whether they are signed or not. Here is a systematic discussion.

2.2.1.1 The long and short of it

In addition to `int`, there are also `short`, `long` and `long long` integers.

- A short int is at least 16 bits;
- An integer is at least 16 bits, which was the case in the old days of the *DEC PDP-11*, but nowadays they are commonly 32 bits;
- A long integer is at least 32 bits, but often 64; the big exception being *Windows* where `long` integers are 32 bits.
- A 'long long' integer is at least 64 bits.
- If you need only one byte for your integer, you can use a `char`; see section **??**.

There are a number of generally accepted *data models* for the definition of these types; see HPC book [11], section 3.7.1.

All these types are signed integers: they accomodate a range $-N \ldots 0 \ldots N+1$ for some N.

If you want to determine precisely what the range of integers or real numbers is that is stored in an `int` variable and such, you can use the `limits` header; see section 3.4.

2.2.1.2 Byte by byte

If you want to dictate how many bits to use, there is the `cstdint` header, which defines such types as `int16_t` and `uint16_t`:

```
int8_t   // 8 bits
uint8_t  // 8 bits, unsigned
int16_t  // 16 bits
uint16_t // 16 bits, unsigned
int64_t  // 64 bits
uint64_t // 64 bits, unsigned
```

Code:

```
1 cout << "max int      : "
2      << numeric_limits<int>::max() <<
       '\n';
3 cout << "max unsigned: "
4      << numeric_limits<unsigned
       int>::max() << '\n';
```

Output
[int] limit:

```
max int     : 2147483647
max unsigned: 4294967295
```

(For the mechanism used here, see section 3.4.)

2.2.1.3 *Integer overflow*

From the limited space that an integer takes, it is clear that there have to be a smallest and largest integer. Querying such limits on integers is discussed in section 3.4.

Computations that exceed those limits have an undefined result; however, since C++20 integers are guaranteed to be stored as 'two's complement'; see HPC book [11], section 3.2.

2.2.1.4 *The dangers of unsigned integers*

Unsigned values are fraught with danger. For instance, comparing them to integers gives counter-intuitive results:

Code:

```
unsigned int one{1};
int mone{-1};
cout << "less: " << boolalpha <<
     (mone<one) << '\n';
```

Output
[int] cmp:

```
less: false
```

For this reason, C++20 has introduced utility functions `cmp_equal` and such (in the `utility` header) that do these comparisons correctly.

Remark 1 *For querying the limits on integers, see section 3.4*

2.2.2 The loop index

2.2.3 Loop index type

In the indexed loops above we use `int` as the type of the loop variable:

```
for ( int i=0; i<some_array.size(); ++i ) { /* stuff */ }
```

There is a problem with this: integers have a maximal value of about 2 billion (see section 3.4 for details), and containers such as `vector` can have more elements than that.

If you absolute need that index variable, give it a type of `size_t`:

```
for ( size_t i=0; i<some_array.size(); ++i ) { /* stuff */ }
```

Alternatively, do not use a loop index, but use a range-based loop, or a range-based algorithm.

Remark 2 *Some compilers will indeed issue a warning on the first loop type, but that is not related to the choice of integer type. Instead, the warning will relate to the fact that in* `i<some_array.size()` *you are comparing a signed to an unsigned quantity. That is considered an unsafe practice.*

2.3 Complex numbers

Complex numbers require the `complex` header. The `complex` type uses templating to set the precision.

```
#include <complex>
complex<float> f;
f.re = 1.; f.im = 2.;

complex<double> d(1.,3.);
```

Math operator like `+`, `*` are defined, as are math functions such as `exp`. Expressions involving a complex number and a simple scalar are well-defined if the scalar is of the underlying type of the complex number:

```
complex<float> x;
x + 1.f; // Yes
x + x.;  // No, because '1.' is double
```

Imaginary unit number i through literals `i`, `if` (float), `il` (long):

```
using namespace std::complex_literals;
std::complex<double> c = 1.0 + 1i;
```

Beware: `1+1i` does not compile.

Code:

```
1 vector< complex<double> > vec1(N, 1.+2.5i
    );
2 auto vec2( vec1 );
3   /* ... */
4 for ( int i=0; i<vec1.size(); ++i ) {
5   vec2[i] = vec1[i] * ( 1.+1.i );
6 }
7   /* ... */
8 auto sum = accumulate
9   ( vec2.begin(),vec2.end(),
10    complex<double>(0.) );
11 cout << "result: " << sum << '\n';
```

Output

```
[complex] vec:
result: (-1.5e+06,3.5e+06)
```

Support:

```
std::complex<T> conj( const std::complex<T>& z );
std::complex<T> exp( const std::complex<T>& z );
```

2.4 Enums and enum classes

The C-style `enum` keyword introduced global names, so

```
enum colors { red,yellow,green };
cout << red << "," << yellow << "," << green << '\n';
enum flag { red,white,blue }; // Collision!
```

does not work.

In C++ the `enum class` (or `enum struct`) was introduced, which makes the names into class members:

```
enum class colors { red,yellow,green };
cout << static_cast<int>( colors::red ) << ","
     << static_cast<int>( colors::yellow ) << ","
     << static_cast<int>( colors::green ) << '\n';
```

If such a class inherits from an integral type, this does not mean it behaves like an integer; for instance, you can not immediately ask if one is less than another. Instead, it only determines the amount of space taken for an enum item.

To let it behave like an integral type,you need to cast it:

```
enum class flag : unsigned int { red,white,blue };
// but we still need to cast them
cout << static_cast<int>( flag::red ) << ","
     << static_cast<int>( flag::white ) << ","
     << static_cast<int>( flag::blue ) << '\n';
```

If you only want a namespace-d enum:

```
class Field {
public:
  enum color { invalid=-1,white=0,red=1,blue=2,green=3 };
private:
  color mycolor;
public:
  void set_color( color c ) { mycolor = c; };
    /* ... */
  Field onefield;
  onefield.set_color( Field::color::blue );
```

2.5 The spaceship operator

It is possible to overload comparison operators `<`, `<=`, `==`, `>=`, `>`, `!=` for classes; section **??**. However, there is usually a lot of redundancy in doing so, even if you express one operator in terms of another:

```
operator>=( const MyClass& other ) {
return not (other<*this);
};
```

The C++20 *spaceship oeprator* `<=>` can make life a lot easier. This operator returns an ordering object, which can be one of the six relations.

Let's illustrate with a simple example. Each object of the class *Record* has a string and a unique number:

```
class Record {
private:
  static inline int nrecords{0};
  int myrecord{-1};
  string name;
public:
  Record( string name )
    : name(name) { myrecord = nrecords++; };
```

Now you want an ordering relation purely based on the record number.

Code:

```
1 Record alice("alice"), bob("bob");
2 cout << "expect  t f  t t  f f\n";
3 cout << boolalpha << (alice==alice) <<
      '\n';
4 cout << boolalpha << (alice==bob) <<
      '\n';
5 cout << boolalpha << (alice<=bob) <<
      '\n';
6 cout << boolalpha << (alice<bob) <<
      '\n';
7 cout << boolalpha << (alice>=bob) <<
      '\n';
8 cout << boolalpha << (alice>bob) <<
      '\n';
```

Output
[stl] spacerecord:

```
expect  t f  t t  f f
true
false
true
true
false
false
```

For this we implement the spaceship operator, using the fact that an ordering on integers exists:

```
std::strong_ordering operator<=>( const Record& other ) {
  return myrecord<=>other.myrecord;
};
bool operator==( const Record& other) {
  return myrecord==other.myrecord;
};
```

Note that the return type of this comparison is `std::strong_ordering`, because in this case any two objects x, y always satisfy exactly one of

```
x<y  x==y  x>y
```

For a more complicated example, let's make a `Coordinate` class, where two coordinates compare as $<, =, >$ if all components satisfy that relation. This is a partial ordering, meaning that some coordinates x, y do not satisfy any of

```
x<y, x==y, x>y
```

Code:

```
1 Coordinate<int>
      p12(1,2),p24(2,4),p31(3,1);
2 cout << "expect t t f\n";
3 cout << boolalpha << (p12==p12) << '\n';
4 cout << boolalpha << (p12<p24) << '\n';
5 cout << boolalpha << (p12<p31 or
      p12>p31 or p12==p31) << '\n';
```

Output
[stl] spacepartial:

```
expect t t f
true
true
false
```

In this case the spaceship operator returns a `partial_ordering` result. Also, we need to define equality separately.

```
std::partial_ordering operator<=>( const Coordinate& other ) const {
  std::strong_ordering c = the_array[0] <=> other.the_array[0];
  for (int i = 1; i <the_array.size(); ++i) {
```

```
      if ((the_array[i] <=> other.the_array[i]) != c)
        return std::partial_ordering::unordered;
    }
    return c;
  };
  bool operator==( const Coordinate& other ) const {
    for ( int ic=0; ic<the_array.size(); ++ic )
      if (the_array[ic]!=other.the_array[ic])
        return false;
    return true;
  };
```

2.6 Some features you may be unaware of

2.6.1 Initializer statements

The C++17 standard introduced a new form of the `if` and `switch` statement: it is possible to have a single statement of declaration prior to the test. This is called the *initializer*.

Code:

```
1 if ( char c = getchar(); c!='a' )
2   cout << "Not an a, but: " << c
3        << '\n';
4 else
5   cout << "That was an a!"
6        << '\n';
```

Output

[basic] ifinit:

```
for c in d b a z ; do \
           echo $c |
    ./ifinit ; \
        done
Not an a, but: d
Not an a, but: b
That was an a!
Not an a, but: z
```

This is particularly elegant if the init statement is a declaration, because the declared variable is then local to the conditional. Previously one would have had to write

```
char c;
c = getchar();
if ( c!='a' ) /* ... */
```

with the variable defined outside of the scope of the conditional.

2.6.2 Const everything!

A const method and its non-const variant are different enough that you can use this for overloading.

Code:

```
1  class has_array {
2  private:
3    vector<float> values;;
4  public:
5    has_array(int l,float v)
6      : values(vector<float>(l,v)) {};
7    auto& at(int i) {
8      cout << "var at" << '\n';
9      return values.at(i); };
10   const auto& at (int i) const {
11     cout << "const at" << '\n';
12     return values.at(i); };
13   auto sum() const {
14     float p;
15     for ( int i=0; i<values.size(); ++i)
16       p += at(i);
17     return p;
18   };
19 };
20
21 int main() {
22
23   int l; float v;
24   cin >> l; cin >> v;
25   has_array fives(l,v);
26   cout << fives.sum() << '\n';
27   fives.at(0) = 2;
28   cout << fives.sum() << '\n';
```

Output
[const] constat:

```
const at
const at
const at
1.5
var at
const at
const at
const at
4.5
```

Exercise 2.1. Explore variations on this example, and see which ones work and which ones not.

1. Remove the second definition of `at`. Can you explain the error?
2. Remove either of the `const` keywords from the second `at` method. What errors do you get?

2.6.3 Mutable

Typical class with non-const update methods, and const readout of some computed quantity:

```
class Stuff {
private:
int i,j;
public:
Stuff(int i,int j) : i(i),j(j) {};
void seti(int inew) { i = inew; };
void setj(int jnew) { j = jnew; };
int result () const { return i+j; };
```

Attempt at caching the computation:

```
class Stuff {
private:
int i,j;
int cache;
void compute_cache() { cache = i+j; };
```

```
public:
Stuff(int i,int j) : i(i),j(j) {};
void seti(int inew) { i = inew; compute_cache(); };
void setj(int jnew) { j = jnew; compute_cache(); };
int result () const { return cache; };
```

But what if setting happens way more often than getting?

```
class Stuff {
private:
int i,j;
int cache;
bool cache_valid{false};
void update_cache() {
if (!cache_valid) {
cache = i+j; cache_valid = true;
};
public:
Stuff(int i,int j) : i(i),j(j) {};
void seti(int inew) { i = inew; cache_valid = false; };
void setj(int jnew) { j = jnew; cache_valid = false; }
int result () const {
update_cache(); return cache; };
```

This does not compile, because `result` is const, but it calls a non-const function.

We can solve this by declaring the cache variables to be **`mutable`**. Then the methods that conceptually don't change the object can still stay **`const`**, even while altering the state of the object. (It is better not to use **`const_cast`**.)

```
class Stuff {
private:
int i,j;
mutable int cache;
mutable bool cache_valid{false};
void update_cache() const {
if (!cache_valid) {
cache = i+j; cache_valid = true;
};
public:
Stuff(int i,int j) : i(i),j(j) {};
void seti(int inew) { i = inew; cache_valid = false; };
void setj(int jnew) { j = jnew; cache_valid = false; }
int result () const {
update_cache(); return cache; };
```

Chapter 3

Floating point arithmetic

3.1 Introduction

In the mathematical sciences, we usually work with real numbers, so it's convenient to pretend that computers can do this too. However, since numbers in a computer have only a finite number of bits, most real numbers can not be represented exactly. In fact, even many fractions can not be represented exactly, since they repeat; for instance, $1/3 = 0.333\ldots$, which is not representable in either decimal or binary.

Real numbers are stored using a scheme that is analogous to what is known as 'scientific notation', where a number is represented as a *significand* and an *exponent*, for instance $6.022 \cdot 10^{23}$, which has a significand `6022` with a *radix point* after the first digit, and an exponent `23`. This number stands for

$$6.022 \cdot 10^{23} = \left[6 \times 10^0 + 0 \times 10^{-1} + 2 \times 10^{-2} + 2 \times 10^{-3}\right] \cdot 10^{23}. \tag{3.1}$$

For a general approach, we introduce a *base* β, which is a small integer number, 10 in the preceding example, and 2 in computer numbers. With this, we write numbers as a sum of t terms:

$$\begin{aligned} x &= \pm 1 \times \left[d_1\beta^0 + d_2\beta^{-1} + d_3\beta^{-2} + \cdots + d_t\beta^{-t+1}b\right] \times \beta^e \\ &= \pm\Sigma_{i=1}^{t} d_i\beta^{1-i} \times \beta^e \end{aligned} \tag{3.2}$$

where the components are

- the *sign bit*: a single bit storing whether the number is positive or negative;
- β is the base of the number system;
- $0 \leq d_i \leq \beta - 1$ the digits of the *mantissa* or *significand* – the location of the radix point (decimal point in decimal numbers) is implicitly assumed to the immediately following the first digit;
- t is the length of the mantissa;
- $e \in [L, U]$ exponent; typically $L < 0 < U$ and $L \approx -U$.

Note that there is an explicit sign bit for the whole number, but the sign of the exponent is handled differently. For reasons of efficiency, e is not a signed number; instead it is considered as an unsigned number in *excess* of a certain minimum value. For instance, with 8 bits for the exponent we can represent the numbers $0 \cdots 256$, but with an excess of 128 this is interpreted as $-127 \cdots 128$. Additionally, the highest and lowest exponent values can be used to express special values such as infinity.

The general definition of floating point numbers, equation (3.2), leaves us with the problem that numbers can have more than one representation. For instance, $.5 \times 10^2 = .05 \times 10^3$. Since this would make computer arithmetic needlessly complicated, for instance in testing equality of numbers, we use *normalized*

floating point numbers. A number is normalized if its first digit is nonzero. The implies that the mantissa part is $1 \leq x_m < \beta$.

A practical implication in the case of binary numbers is that the first digit is always 1, so we do not need to store it explicitly. In the *IEEE 754* standard, this means that every normalized floating point number is of the form

$$1.d_1 d_2 \ldots d_t \times 2^e \tag{3.3}$$

and only the digits $d_1 d_2 \ldots d_t$ are stored.

Exercise 3.1. With quantities t, e, what is the smallest positive normalized number?

Unnormalized numbers do exist, but only for numbers smaller than the smallest positive normalized number. See section 3.2.2.

3.2 IEEE 754

Some decades ago, issues such as the length of the mantissa and the rounding behavior of operations could differ between computer manufacturers, and even between models from one manufacturer. This was obviously a bad situation from a point of portability of codes and reproducibility of results. The *IEEE*

sign	exponent	mantissa
p	$e = e_1 \cdots e_8$	$s = s_1 \ldots s_{23}$
31	$30 \cdots 23$	$22 \cdots 0$
$\pm$	2^{e-127} (except $e = 0, 255$)	$2^{-s_1} + \cdots + 2^{-s_{23}}$

sign	exponent	mantissa
s	$e_1 \cdots e_{11}$	$s_1 \ldots s_{52}$
63	$62 \cdots 52$	$51 \cdots 0$
$\pm$	2^{e-1023} (except $e = 0, 2047$)	$2^{-s_1} + \cdots + 2^{-s_{52}}$

(3.4)

Figure 3.1: Single and double precision definition.

754 standard [1] codified all this in 1985, for instance stipulating 24 and 53 bits (before normalization) for the mantissa in single and double precision arithmetic, using a storage sequence of sign bit, exponent, mantissa; see figure 3.1.

Remark 3 *The full name of the 754 standard is 'IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Std 754-1985)'. It is also identical to IEC 559: 'Binary floating-point arithmetic for microprocessor systems', superseded by ISO/IEC/IEEE 60559:2011.*

IEEE 754 is a standard for binary arithmetic; there is a further standard, IEEE 854, *that allows decimal arithmetic.*

Remark 4 *'It was remarkable that so many hardware people there, knowing how difficult p754 would be, agreed that it should benefit the community at large. If it encouraged the production of floating-point software and eased the development of reliable software, it would help create a larger market for everyone's hardware. This degree of altruism was so astonishing that MATLAB's creator Dr. Cleve Moler used to advise foreign visitors not to miss the country's two most awesome spectacles: the Grand Canyon, and meetings of IEEE p754.' W. Kahan,* `http://www.cs.berkeley.edu/~wkahan/ieee754status/754story.html`.

The standard also declared the rounding behavior to be *correct rounding*: the result of an operation should be the rounded version of the exact result. See section 3.2.3.

exponent	numerical value	range
$(0\cdots0)=0$	$\pm 0.s_1\cdots s_{23}\times 2^{-126}$	$s=0\cdots01 \Rightarrow 2^{-23}\cdot 2^{-126}=2^{-149}\approx 10^{-50}$ $s=1\cdots11 \Rightarrow (1-2^{-23})\cdot 2^{-126}$
$(0\cdots01)=1$	$\pm 1.s_1\cdots s_{23}\times 2^{-126}$	$s=0\cdots00 \Rightarrow 1\cdot 2^{-126}\approx 10^{-37}$
$(0\cdots010)=2$	$\pm 1.s_1\cdots s_{23}\times 2^{-125}$	
$\cdots$		
$(01111111)=127$	$\pm 1.s_1\cdots s_{23}\times 2^{0}$	$s=0\cdots00 \Rightarrow 1\cdot 2^{0}=1$ $s=0\cdots01 \Rightarrow 1+2^{-23}\cdot 2^{0}=1+\epsilon$ $s=1\cdots11 \Rightarrow (2-2^{-23})\cdot 2^{0}=2-\epsilon$
$(10000000)=128$	$\pm 1.s_1\cdots s_{23}\times 2^{1}$	
$\cdots$		
$(11111110)=254$	$\pm 1.s_1\cdots s_{23}\times 2^{127}$	
$(11111111)=255$	$s_1\cdots s_{23}=0 \Rightarrow \pm\infty$ $s_1\cdots s_{23}\neq 0 \Rightarrow$ `NaN`	

(3.5)

Figure 3.2: Interpretation of single precision numbers depending on the exponent bit pattern.

An inventory of the meaning of all bit patterns in IEEE 754 single precision is given in figure 3.2. Recall from section 3.1 above that for normalized numbers the first nonzero digit is a 1, which is not stored, so the bit pattern $d_1 d_2 \ldots d_t$ is interpreted as $1.d_1 d_2 \ldots d_t$.

The highest exponent is used to accommodate `Inf` and `Nan`; see section **??**.

Exercise 3.2. Every programmer, at some point in their life, makes the mistake of storing a real number in an integer or the other way around. This can happen for instance if you call a function differently from how it was defined.

```
void a(double x) {....}
int main() {
int i;
.... a(i) ....
```

```
}
```

What happens when you print `x` in the function? Consider the bit pattern for a small integer, and use the table in figure 3.2 to interpret it as a floating point number. Explain that it will be a subnormal number.

(This is one of those errors you won't forget after you make it. In the future, whenever you see a number on the order of 10^{-305} you'll recognize that you probably made this error.)

These days, almost all processors adhere to the IEEE 754 standard. Early generations of the *NVidia Tesla Graphics Processing Unit (GPU)*s were not standard-conforming in single precision. The justification for this was that single precision is more likely used for graphics, where exact compliance matters less. For many scientific computations, double precision is necessary, since the precision of calculations gets worse with increasing problem size or runtime. This is true for the sort of calculations in chapter **??**, but not for others such as the *Lattice Boltzmann Method (LBM)*.

3.2.1 Machine precision

Often we are only interested in the order of magnitude of the representation error, and we will write $\tilde{x} = x(1+\epsilon)$, where $|\epsilon| \leq \beta^{-t}$. This maximum relative representation error is called the *machine precision*, or sometimes *machine epsilon*. Typical values are:

$$\begin{cases} \epsilon \approx 10^{-7} & \text{32-bit single precision} \\ \epsilon \approx 10^{-16} & \text{64-bit double precision} \end{cases} \tag{3.6}$$

(After you have studied the section on the IEEE 754 standard, can you derive these values?)

Machine precision can be defined another way: ϵ is the smallest number that can be added to 1 so that $1+\epsilon$ has a different representation than 1. A small example shows how aligning exponents can shift a too small operand so that it is effectively ignored in the addition operation:

$$\begin{array}{rll} & 1.0000 & \times 10^{0} \\ + & 1.0000 & \times 10^{-5} \\ \hline \end{array} \quad \Rightarrow \quad \begin{array}{rll} & 1.0000 & \times 10^{0} \\ + & 0.00001 & \times 10^{0} \\ \hline = & 1.0000 & \times 10^{0} \end{array} \tag{3.7}$$

Yet another way of looking at this is to observe that, in the addition $x+y$, if the ratio of x and y is too large, the result will be identical to x.

The machine precision is the maximum attainable accuracy of computations: it does not make sense to ask for more than 6-or-so digits accuracy in single precision, or 15 in double.

Exercise 3.3. Write a small program that computes the machine epsilon. Does it make any difference if you set the *compiler optimization levels* low or high? (If you speak C++, can you solve this exercise with a single templated code?)

Exercise 3.4. The number $e \approx 2.72$, the base for the natural logarithm, has various definitions. One of them is

$$e = \lim_{n \to \infty} (1 + 1/n)^n. \tag{3.8}$$

Write a single precision program that tries to compute e in this manner. (Do not use the `pow` function: code the power explicitly.) Evaluate the expression for an upper bound $n = 10^k$ for some k. (How far do you let k range?) Explain the output for large n. Comment on the behavior of the error.

Exercise 3.5. The exponential function e^x can be computed as

$$e = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots \tag{3.9}$$

Code this and try it for some positive x, for instance $x = 1, 3, 10, 50$. How many terms do you compute?

Now compute e^{-x} for those values. Use for instance the same number of iterations as for e^x.

What do you observe about the e^x versus e^{-x} computation? Explain.

3.2.2 Underflow

The normalized number closest to zero is $1 \cdot \beta^L$.

	unit	fractional			exponent
digit	1	0	$\cdots$	0	
value	1	0	$\cdots$	0	L

(3.10)

Trying to compute a number less than that in absolute value is sometimes handled by using *subnormal* (or *denormalized* or *unnormalized*) numbers, a process known as *gradual underflow*. In this case, a special value of the exponent indicates that the number is no longer normalized. In the case IEEE standard arithmetic (section 3.2) this is done through a zero exponent field.

However, this is typically tens or hundreds of times slower than computing with regular floating point numbers . At the time of this writing, only the IBM Power6 (and up) has hardware support for gradual underflow. Section **??** explores performance implications.

3.2.2.1 What happens with over/underflow?

The occurrence of overflow or underflow means that your computation will be 'wrong' from that point on. Underflow will make the computation proceed with a zero where there should have been a nonzero; overflow is represented as `Inf`, short for 'infinite'.

Exercise 3.6. For real numbers x, y, the quantity $g = \sqrt{(x^2 + y^2)/2}$ satisfies

$$g \leq \max\{|x|, |y|\} \tag{3.11}$$

so it is representable if x and y are. What can go wrong if you compute g using the above formula? Can you think of a better way?

Computing with `Inf` is possible to an extent: adding two of those quantities will again give `Inf`. However, subtracting them gives `NaN`: 'not a number'. (See section **??**.)

In none of these cases will the computation end: the processor will continue, unless you tell it otherwise. The 'otherwise' consists of you telling the compiler to generate an *interrupt*, which halts the computation with an error message. See section **??**.

3.2.3 Correct rounding

The IEEE 754 standard, mentioned in section 3.2, does not only declare the way a floating point number is stored, it also gives a standard for the accuracy of operations such as addition, subtraction, multiplication, division. The model for arithmetic in the standard is that of *correct rounding*: the result of an operation should be as if the following procedure is followed:

- The exact result of the operation is computed, whether this is representable or not;
- This result is then rounded to the nearest computer number.

In short: the representation of the result of an operation is the rounded exact result of that operation. Of course, after two operations it no longer needs to hold that the computed result is the exact rounded version of the exact result.

If this statement sounds trivial or self-evident, consider subtraction as an example. In a decimal number system with two digits in the mantissa, the computation

$$\begin{aligned} 1.0 - 9.4 \cdot 10^{-1} = & \\ 1.0 - 0.94 &= 0.06 \\ &= 0.6 \cdot 10^{-2} \end{aligned} \tag{3.12}$$

Note that in an intermediate step the mantissa .094 appears, which has one more digit than the two we declared for our number system. The extra digit is called a *guard digit*.

Without a guard digit, this operation would have proceeded as $1.0 - 9.4 \cdot 10^{-1}$, where $9.4 \cdot 10^{-1}$ would be normalized to 0.9, giving a final result of 0.1, which is almost double the correct result.

Exercise 3.7. Consider the computation $1.0 - 9.5 \cdot 10^{-1}$, and assume again that numbers are rounded to fit the 2-digit mantissa. Why is this computation in a way a lot worse than the example?

One guard digit is not enough to guarantee correct rounding. An analysis that we will not reproduce here shows that three extra bits are needed [12].

3.3 Why it is important to know this

3.3.1 Associativity

Another effect of the way floating point numbers are treated is on the *associativity* of operations such as summation. While summation is mathematically associative, this is no longer the case in computer arithmetic.

Let's consider a simple example, showing how associativity is affected by the rounding behavior of floating point numbers. Let floating point numbers be stored as a single digit for the mantissa, one digit

for the exponent, and one guard digit; now consider the computation of $4+6+7$. Evaluation left-to-right gives:

$$\begin{aligned}
(4 \cdot 10^0 + 6 \cdot 10^0) + 7 \cdot 10^0 &\Rightarrow 10 \cdot 10^0 + 7 \cdot 10^0 && \text{addition} \\
&\Rightarrow 1 \cdot 10^1 + 7 \cdot 10^0 && \text{rounding} \\
&\Rightarrow 1.0 \cdot 10^1 + 0.7 \cdot 10^1 && \text{using guard digit} \\
&\Rightarrow 1.7 \cdot 10^1 \\
&\Rightarrow 2 \cdot 10^1 && \text{rounding}
\end{aligned} \tag{3.13}$$

On the other hand, evaluation right-to-left gives:

$$\begin{aligned}
4 \cdot 10^0 + (6 \cdot 10^0 + 7 \cdot 10^0) &\Rightarrow 4 \cdot 10^0 + 13 \cdot 10^0 && \text{addition} \\
&\Rightarrow 4 \cdot 10^0 + 1 \cdot 10^1 && \text{rounding} \\
&\Rightarrow 0.4 \cdot 10^1 + 1.0 \cdot 10^1 && \text{using guard digit} \\
&\Rightarrow 1.4 \cdot 10^1 \\
&\Rightarrow 1 \cdot 10^1 && \text{rounding}
\end{aligned} \tag{3.14}$$

The conclusion is that the sequence in which rounding and truncation is applied to intermediate results makes a difference.

Exercise 3.8. The above example used rounding. Can you come up with a similar example in an arithmetic system using truncation?

3.3.2 Practical example: the 'abc' formula

As a practical example, consider the quadratic equation $ax^2 + bx + c = 0$ which has solutions $x = \frac{-b \pm \sqrt{b^2-4ac}}{2a}$. Suppose $b > 0$ and $b^2 \gg 4ac$ then $\sqrt{b^2 - 4ac} \approx b$ and the '$+$' solution will be inaccurate. In this case it is better to compute $x_- = \frac{-b-\sqrt{b^2-4ac}}{2a}$ and use $x_+ \cdot x_- = c/a$.

Exercise 3.9. Explore computing the roots of

$$\epsilon x^2 - (1 + \epsilon^2)x + \epsilon \tag{3.15}$$

by the 'textbook' method and as described above.

- What are the roots?
- Why does the textbook method compute one root as zero, for ϵ small enough? Can you think of an example where this is very bad?
- What are the computed function values in both methods? Relative errors?

Exercise 3.10. Write a program that computes the roots of the quadratic equation, both the 'textbook' way, and as described above.

- Let $b = -1$ and $a = -c$, and $4ac \downarrow 0$ by taking progressively smaller values for a and c.
- Print out the computed root, the root using the stable computation, and the value of $f(x) = ax^2 + bx + c$ in the computed root.

Now suppose that you don't care much about the actual value of the root: you want to make sure the residual $f(x)$ is small in the computed root. Let x^* be the exact root, then

$$f(x^* + h) \approx f(x^*) + hf'(x^*) = hf'(x^*). \tag{3.16}$$

Now investigate separately the cases $a \downarrow 0, c = -1$ and $a = -1, c \downarrow 0$. Can you explain the difference?

Exercise 3.11. Consider the functions

$$\begin{cases} f(x) = \sqrt{x+1} - \sqrt{x} \\ g(x) = 1/\left(\sqrt{x+1} + \sqrt{x}\right) \end{cases} \tag{3.17}$$

- Show that they are the same in exact arithmetic; however:
- Show that f can exhibit cancellation and that g has no such problem.
- Write code to show the difference between f and g. You may have to use large values for x.
- Analyze the cancellation in terms of x and machine precision. When are $\sqrt{x+1}$ and $\sqrt{x}$ less than ϵ apart? What happens then? (For a more refined analysis, when are they $\sqrt{\epsilon}$ apart, and how does that manifest itself?)
- The inverse function of $y = f(x)$ is

$$x = (y^2 - 1)^2/(4y^2) \tag{3.18}$$

  Add this to your code. Does this give any indication of the accuracy of the calculations?

Make sure to test your code in single and double precision. If you speak python, try the `bigfloat` package.

3.4 Limits

There used to be a header file `limits.h` that contained macros such as `MAX_INT` and `MIN_INT`. While this is still available, the Standard Template Library (STL) offers a better solution in the `numeric_limits` function of the `numeric` header.

Use header file `limits`:

```
#include <limits>
using std::numeric_limits;

cout << numeric_limits<long>::max();
```

- The largest number is given by `max`; use `lowest` for 'most negative'.
- The smallest denormal number is given by `denorm_min`.
- `min` is the smallest positive number that is not a denormal;
- There is an `epsilon` function for machine precision:

Code:

```
cout << "Single lowest "
     << numeric_limits<float>::lowest()
     << " and epsilon "
     << numeric_limits<float>::epsilon()
     << '\n';
cout << "Double lowest "
     << numeric_limits<double>::lowest()
     << " and epsilon "
     << numeric_limits<double>::epsilon()
     << '\n';
```

Output
[stl] eps:

```
Single lowest -3.40282e+38 and
    epsilon 1.19209e-07
Double lowest -1.79769e+308 and
    epsilon 2.22045e-16
```

Code:

```
cout << "Signed int: "
     << numeric_limits<int>::min() << " "
     << numeric_limits<int>::max()
     << '\n';
cout << "Unsigned    "
     << numeric_limits<unsigned int>::min() <<
    " "
     << numeric_limits<unsigned int>::max()
     << '\n';
cout << "Single      "
     << numeric_limits<float>::denorm_min() <<
    " "
     << numeric_limits<float>::min() << " "
     << numeric_limits<float>::max()
     << '\n';
cout << "Double      "
     << numeric_limits<double>::denorm_min()
    << " "
     << numeric_limits<double>::min() << " "
     << numeric_limits<double>::max()
     << '\n';
```

Output
[stl] limits:

```
Signed int: -2147483648
    2147483647
Unsigned    0 4294967295
Single      1.4013e-45
    1.17549e-38 3.40282e+38
Double      4.94066e-324
    2.22507e-308 1.79769e+308
```

Exercise 3.12. Write a program to discover what the maximal n is so that $n!$, that is, n-factorial, can be represented in an `int`, `long`, or `long long`. Can you write this as a templated function?

Operations such as dividing by zero lead to floating point numbers that do not have a valid value. For efficiency of computation, the processor will compute with these as if they are any other floating point number.

Chapter 4

Structured programming

4.1 Modules

The C++20 standard is taking a different approach to header files, through the introduction of *modules*. This largely dispenses with the need for header files included through the C Preprocessor (CPP). However, the CPP may still be needed for other purposes.

4.1.1 Program structure with modules

Using modules, the `#include` directive is no longer needed; instead, the `import` keyword indicates what module is to be used in a (sub)program:

```
import fg_module;
int main() {
  std::cout << "Hello world " << f(5) << '\n';
```

The module is in a different file; the `export` keyword, followed by `module` defines the name of the module. This file can then have any number of functions; only the ones with the `export` keyword will be visible in a program that imports the module.

```
export module fg_module;

// internal function
int g( int i ) { return i/2; };

// exported function
export int f( int i ) {
  return g(i+1);
};
```

4.1.2 Implementation and interface units

A module can have a leveled structure, by using names with a `module:partition` structure.

This makes it possible to have separate

- Interface partitions, that define the interface to the using program; and
- Implementation partitions, that contain the code that needs to be shielded from the user.

Here is an implementation partition; there is no `import` keyword because this functionality is internal:

```
// implementation unit, nothing exported
module helper_module:helper;
// internal function
int g( int i ) { return i/2; };
```

Here is an interface partition, which uses the internal function, and exports a different function:

```
export module helper_module;
import :helper;

// exported function
export int f( int i ) {
  return g(i+1);
};
```

4.2 Namespaces

4.3 Solving name conflicts

In section **??** you saw that the C++ STL 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 **using namespace**. You put

```
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 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.

You can make your own namespace by writing

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

so that you can write

Qualify type with namespace:

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

or

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

or

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

or

```
using namespace abc = space_a::space_b::space_c;
abc::func(x)
```

4.3.1 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:

There is a `vector` in the standard namespace and in the new `geometry` namespace:

```
#include <vector>
#include "geolib.hpp"
using namespace geometry;
int main() {
  std::vector< vector > vectors;
  vectors.push_back( vector( point(1,1),point(4,5) ) );
```

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 );
  };
}
```

4.4 Namespaces and libraries

As the introduction to this chapter argued, namespaces are a good way to prevent name conflicts. This means that it's a good idea to create a namespace for all your routines. You see this design in almost all published C++ libraries.

Now consider this scenario:

1. You write a program that uses a certain library;
2. A new version of the library is released and installed on your system;
3. Your program, using shared/dynamic libraries, starts using this library, maybe even without you realizing it.

This means that the old and new libraries need to be compatible in several ways:

1. All the classes, functions, and data structures defined in the earlier version also need to be defined in the new. This is not a big problem: new library versions typically add functionality. However,
2. The data layout of the new version needs to be the same.

That second point is subtle. To illustrate, consider the library is upgraded:

First version:

```
}

namespace geometry {
  class vector {
    private:
    std::vector<float> data;
    std::string name;
  }
```

New version:

```
namespace geometry {
  class vector {
    private:
    std::vector<float> data;
    int id;
    std::string name;
  }
}
```

The problem is that your compiled program has explicit information where the class members are located in the class object. By changing that structure of the objects, those references are no longer correct. This is called 'Application Binary Interface (ABI) breakage' and it leads to *undefined behavior*.

The library can prevent this by:

```
namespace geometry {
  inline namespace v1.0 {
    class vector {
      private:
      std::vector<float> data;
      std::string name;
    }
  }
}
```

and updating the version number in future version. The program using this library implicitly uses the namespace `geometry::v1.0::vector` so after a library update, trying to execute the program

4.5 Best practices

In this course we advocated pulling in functions explicitly:

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

It is also possible to use

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

The problem with this is that it may pull in unwanted functions. For instance:

This compiles, but should not:

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

def swop(int i,int j) {};

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

This gives an error:

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

def swop(int i,int j) {};

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

Even if you use `using namespace`, you only do this in a source file, not in a header file. Anyone using the header would have no idea what functions are suddenly defined.

Chapter 5

Include files

The `#include` pragma causes the named file to be included. That file is in fact an ordinary text file, stored somewhere in your system. As a result, your source file is transformed to another source file, in a source-to-source translation stage, and only that second file is actually compiled by the *compiler*.

Normally, this intermediate file with all included literally included is immediately destroyed again, but in rare cases you may want to dump it for debugging. See your compiler documentation for how to generate this file.

5.1 Kinds of includes

While you can include any kind of text file, the typical use of the preprocessor is to include a *header file* at the start of your source.

There are two kinds of includes

1. The file name can be included in angle brackets,

   ```
   #include <vector>
   ```

   which is typically used for system headers that are part of the compiler infrastructure;
2. The name can also be in double quotes,

   ```
   #include "somelib.h"
   ```

   which is typically used for files that are part of your code, or for libraries that you have downloaded and installed yourself.

5.2 Search paths

System headers can usually be found by the compiler because they are in some standard location. Including other files may require additional action by you. If you write

```
#include "foo.h"
```

the compiler only looks for that file in the current directory where you are compiling.

If the include file is part of a library that you have downloaded and installed yourself, say it is in

```
/home/yourname/mylibrary/include/foo.h
```

then you could of course spell out the location:

```
#include "/home/yourname/mylibrary/include/foo.h"
```

However, that does not make your code very portable to other systems and other users. So how do you make

```
#include "foo.h"
```

be understood on any system?

The answer is that you can give your compiler one or more *include path*. This is done with the `-I` flag.

```
icpc -c yourprogram.cpp -I/home/yourname/mylibrary/include
```

You can specify more than one such flag, and the compiler will try to find the `foo.h` file in all of them.

Are you now thinking that you have to type that path every time you compile? This is the time to learn about the utilities *Make* Tutorials book [9], chapter-3 or *CMake* Tutorials book [9], chapter-4.

Chapter 6

Arrays

6.1 Wrapping an array in a class

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

```
class namedvector {
private:
  string name;
  vector<int> values;
public:
  namedvector(int n,string name="unnamed")
    : name(name),values(vector<int>(n)) {
  };
  string rendered() {
    stringstream render;
    render << name << ":";
    for (auto v : values )
      render <<  " " << v << ",";
    return render.str();
  }
    /* ... */
};
```

Unfortunately this means you may have to recreate some methods:

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

Another way out of this is to inherit from a container class.

6.2 Multi-dimensional array

Unlike Fortran, C++ has little support for multi-dimensional arrays. If your multi-dimensional arrays are to model linear algebra objects, it would be good to check out the *Eigen* library. Here are some general remarks on multi-dimensional storage.

6.2.1 Matrix as vector of vectors

Multi-dimensional is harder with vectors:

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

Create a row vector, then store 10 copies of that:
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.

Remark 5 *More flexible strategy:*

```
// alternative:
vector<vector<float>> rows(10);
for ( auto &row : rows )
row = vector<float>(20);
```

```
1  class matrix {
2  private:
3    vector<vector<double>> elements;
4  public:
5    matrix(int m,int n) {
6      elements =
7        vector<vector<double>>(m,vector<double>(n));
8    }
9    void set(int i,int j,double v) {
10     elements.at(i).at(j) = v;
11   };
12   double get(int i,int j) {
13     return elements.at(i).at(j);
14   };
```

Exercise 6.1. Write `rows()` and `cols()` methods for this class that return the number of rows and columns respectively.

Exercise 6.2. Write a method `void set(double)` that sets all matrix elements to the same value.

Write a method `double totalsum()` that returns the sum of all elements.

Code:

```
1 A.set(3.);
2 cout << "Sum of elements: "
3      << A.totalsum() << '\n';
```

Output
[array] matrixsum:

```
Sum of elements: 30
```

You can base this off the file `matrix.cpp` in the repository

Exercise 6.3. Add methods such as `transpose`, `scale` to your matrix class.

Implement matrix-matrix multiplication.

6.2.2 A better matrix class

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

```
// /matrixclass.cpp
class matrix {
private:
  std::vector<double> the_matrix;
  int m,n;
public:
  matrix(int m,int n)
    : m(m),n(n),the_matrix(m*n) {};
  void set(int i,int j,double v) {
    the_matrix.at( i*n +j ) = v;
  };
  double get(int i,int j) {
    return the_matrix.at( i*n +j );
  };
    /* ... */
// /matrixclass.cpp
};
```

Exercise 6.4. In the matrix class of the previous slide, why are m, n stored explicitly, and not in the previous case?

The most important advantage of this is that it is compatible with the storage traditionally used in many libraries and codes.

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

Exercise 6.5. Write a method `element` of type `double&`, so that you can write

```
A.element(2,3) = 7.24;
```

Chapter 7

Input/output

Most programs take some form of input, and produce some form of output. In a beginning course such as this, the output is of more importance than the input, and what output there is only goes to the screen, so that's what we start by focusing on. We will also look at output to file, and input.

7.1 Screen output

In examples so far, you have used `cout` with its default formatting. In this section we look at ways of customizing the `cout` output.

Remark 6 *Even after the material below you may find* `cout` *not particularly elegant. In fact, if you've programmed in C before, you may prefer the* `printf` *mechanism. The C++20 standard has the* `format` *header, which is as powerful as* `printf`*, but considerably more elegant.*

However, as of early 2022 this is not available in most compilers, so in section 7.6 we will give examples from fmtlib*, the open source library that gave rise to* `std::format`*.*

From `iostream`: `cout` uses default formatting.

Possible manipulation in `iomanip` header: pad a number, use limited precision, format as hex, etc.

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

Code:

```
1 for (int i=1; i<200000000; i*=10)
2   cout << "Number: " << i << '\n';
3 cout << '\n';
```

Output
[io] cunformat:

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

We will now look at some examples of non-standard formatting. You may for instance want to output several lines, with the numbers in them nicely aligned. The most common I/O manipulation is to set a uniform width, that is, use the same number of positions for each number, regardless how many they need.

- The `setw` specifies how many positions to use for the following number.
- Note the singular in the previous sentence: the `setw` specifier applies only once.
- By default, numbers are right-aligned in the space given for them, and if they require more positions, they overflow on the right.

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

Code:

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

  // 'setw' applies only once:
  cout << "Width is 6:" << '\n';
  cout << ">"
       << setw(6) << 1 << 2 << 3 << '\n';
  cout << '\n';
```

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
```

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

Code:

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

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.

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 << '\n';
```

Output
[io] formatleft:

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

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 << '\n';
  }
```

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

Exercise 7.1. Make the first line in the above output align better with the other lines:

```
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
```

Hex output is useful for addresses (chapter **??**):

Code:

```
1 int i;
2 cout << "address of i, decimal: "
3      << (long)&i << '\n';
4 cout << "address of i, hex    : "
5      << std::hex << &i << '\n';
```

Output
[pointer] coutpoint:

```
address of i, decimal:
    140732703427524
address of i, hex    :
    0x7ffee2cbcbc4
```

Back to decimal:

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

There is no standard modifier for outputting as binary. However, you can use the `bitset` header to print the bit pattern of an integer.

Code:

```
1 #include <bitset>
2 using std::bitset;
3     /* ... */
4   auto x255 = bitset<16>(255);
5   cout << x255 << '\n';
```

Output
[io] bits:

```
0000000011111111
```

7.1.1 Floating point output

The output of floating point numbers is more tricky.

- How many positions are used for the digits before the decimal point?
- How many digits after the decimal point are printed?
- Is scientific notation used?

For floating point numbers, the `setprecision` modifier determines how many positions are used for the integral and fractional part together. If the integral part takes more positions, scientific notation is used.

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

Code:

```
1 #include <iomanip>
2 using std::left;
3 using std::setfill;
4 using std::setw;
5 using std::setprecision;
6     /* ... */
7   x = 1.234567;
8   for (int i=0; i<10; ++i) {
9     cout << setprecision(4) << x << '\n';
10    x *= 10;
11  }
```

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
```

This mode is a mix of fixed and floating point. See the `scientific` option below for consistent use of floating point format.

With the `fixed` modifier, `setprecision` applies to the fractional part.

Fixed precision applies to fractional part:

Code:

```
x = 1.234567;
cout << fixed;
for (int i=0; i<10; ++i) {
  cout << setprecision(4) << x << '\n';
  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
```

(Notice the rounding)

The `setw` modifier, for fixed point output, applies to the total width of integral and fractional part, plus the decimal point.

Combine width and precision:

Code:

```
x = 1.234567;
cout << fixed;
for (int i=0; i<10; ++i) {
  cout << setw(10) << setprecision(4)
    << x
       << '\n';
  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
```

Exercise 7.2. Use integer output to print real numbers aligned on the decimal:

Code:

```
string quasifix(double);
int main() {
  for ( auto x : { 1.5, 12.32, 123.456,
    1234.5678 } )
    cout << quasifix(x) << '\n';
```

Output
[io] quasifix:

```
   1.5
  12.32
 123.456
1234.5678
```

Use four spaces for both the integer and fractional part; test only with numbers that fit this format.

Above you saw that `setprecision` may give both fixed and floating point output. To get strictly floating point 'scientific' notation output, use `scientific`.

Combining width and precision:

Code:

```
x = 1.234567;
cout << scientific;
for (int i=0; i<10; ++i) {
  cout << setw(10) << setprecision(4)
       << x << '\n';
  x *= 10;
}
cout << '\n';
```

Output
[io] iofsci:

```
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
```

7.1.2 Boolean output

The *boolalpha* modifier renders a **bool** variable as **true**, **false**.

7.1.3 Saving and restoring settings

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

cout.flags(old_settings);

int old_precision = cout.precision();

cout.precision(old_precision);
```

7.2 File output

The *iostream* is just one example of a *stream*, which is a general mechanism for converting entities to exportable form.

In particular, file output works the same as screen output: after you create a stream variable, you can 'lessless' to it.

```
mystream << "x: " << x << '\n';
```

The following example uses an *ofstream*: an output file stream. This has an *open* method to associate it with a file, and a corresponding *close* method.

Use:

Code:

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

Output
[io] fio:

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

Compare: `cout` is a stream that has already been opened to your terminal 'file'.

The *open* call can have flags, for instance for appending:

```
file.open(name,std::fstream::out | std::fstream::app);
g
```

Binary output: write your data byte-by-byte from memory to file.
(Why is that better than a printable representation?)

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
```

7.3 Output your own classes

You have used statements like:

```
cout << "My value is: " << myvalue << "\n";
```

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.

Here we solve this in two steps:

Define a function that yields a string representing the object, and

```
string as_string() {
  stringstream ss;
  ss << "(" << x << "," << y << ")";
  return ss.str();
};
  /* ... */
std::ostream& operator<<
(std::ostream &out,Point &p) {
  out << p.as_string(); return out;
};
```

Redefine the less-less operator to use this.

```
Point p1(1.,2.);
cout << "p1 " << p1
     << " has length "
     << p1.length() << "\n";
```

(See section **??** for *stringstream* and the *sstream* header.)

If you don't want to write that accessor function, you can declare the lessless operator as a **friend**:

```
class container {
private: double x;
public:
  friend ostream& operator<<( ostream& s,const container& c ) {
    s << c.x; /* no accessor */
    return s; };
};
```

7.4 Output buffering

In C, the way to get a newline in your output was to include the character `\n` in the output. This still works in C++, and at first it seems there is no difference with using **endl**. However, **endl** does more than breaking the output line: it performs a **std**::*flush*.

7.4.1 The need for flushing

Output is usually not immediately written to screen or disc or printer: it is saved up in buffers. This can be for efficiency, because output a single character may have a large overhead, or it may be because the device is busy doing something else, and you don't want your program to hang waiting for the device to free up.

However, a problem with buffering is the output on the screen may lag behind the actual state of the program. In particular, if your program crashes before it prints a certain message, does it mean that it crashed before it got to that line, or does it mean that the message is hanging in a buffer.

This sort of output, that absolutely needs to be handled when the statement is called, is often called *logging* output. The fact that **endl** does a flush would mean that it would be good for logging output. However, it also flushes when not strictly necessary. In fact there is a better solution: **std**::*cerr* works just like **cout**, except it doesn't buffer the output.

7.4.2 Performance considerations

If you want a newline in your output (whether screen or more general stream), using `endl` may slow down your program because of the flush it performs. More efficiently, you would add a newline character to the output directly:

```
somestream << "Value: " << x << '\n';
otherstream << "Total " << nerrors << " reported\n";
```

In other words, use `cout` for regular output, `cerr` for logging output, and use `\n` instead of `endl`.

7.5 Input

The `cin` command can be used to read integers and floating point formats.

Code:

```
1 float input;
2 cin >> input;
3 cout << "(I think I got: " << input <<
      ")\n";
```

Output
[io] cinfloat:

```
for n in \
            1.5  1.6  1.67
    1.67e5  2.5.6  \
            ; do \
                echo $n |
    ./cinfloat \
        ; done
(I think I got: 1.5)
(I think I got: 1.6)
(I think I got: 1.67)
(I think I got: 167000)
(I think I got: 2.5)
```

As is illustrated with the last number in this example, `cin` will read until the first character that does not fit the format of the variable, in this case the second period. On the other hand, the `e` in the number before it is interpreted as the exponent of a floating point representation.

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;
#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/.

7.5.1 File input

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 (getline(input_file,oneline)) {
    cout << "Got line: <<" << oneline << ">>" << '\n';
  }
```

There are several ways of testing for the end of a file

- For text files, the *getline* function returns **false** if no line can be read.
- The *eof* function can be used after you have done a read.
- *EOF* is a return code of some library functions; it is not true that a file ends with an EOT character. Likewise you can not assume a Control-D or Control-Z at the end of the file.

Exercise 7.3. 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

Advanced note: You may think that *getline* always returns a **bool**, but that's not true. If actually returns an *ifstream*. However, a conversion operator

```
explicit operator bool() const;
```

exists for anything that inherits from *basic_ios*.

7.5.2 Input streams

Tests, mostly for file streams: *is_eof* *is_open*

7.5.3 C-style file handling

The old *FILE* type should not be used anymore.

7.6 Fmtlib

7.6.1 basics

- *print* for printing,
  format gives **std**::*string*;
- Arguments indicated by curly braces;
- braces can contain numbers (and modifiers, see next)

Code:

```
auto hello_string = fmt::format
  ("{} {}!","Hello","world");
cout << hello_string << '\n';
fmt::print
  ("{0}, {0} {1}!\n","Hello","world");
```

Output
[io] fmtbasic:

```
Hello world!
Hello, Hello world!
```

API documentation: https://fmt.dev/latest/api.html

7.6.2 Align and padding

In `fmtlib`, the 'greater than' sign plus a number indicates right aligning and the width of the field.

Code:

```
for (int i=10; i<2000000000; i*=10)
  fmt::print("{:>6}\n",i);
```

Output
[io] fmtwidth:

```
    10
   100
  1000
 10000
100000
1000000
10000000
100000000
1000000000
1410065408
1215752192
```

Code:

```
for (int i=10; i<2000000000; i*=10)
  fmt::print("{0:.>6}\n",i);
```

Output
[io] fmtleftpad:

```
....10
...100
..1000
.10000
100000
1000000
10000000
100000000
1000000000
1410065408
1215752192
```

7.6.3 Construct a string

If you want to construct a string piecemeal, for instance because it involves a loop over something, you can use a *memory_buffer*:

```
fmt::memory_buffer b;
fmt::format_to(std::back_inserter(b),"[");
for ( auto i : indices )
  fmt::format_to(std::back_inserter(b),"{}, ",i);
fmt::format_to(std::back_inserter(b),"]");
cout << to_string(b) << endl;
```

7.6.4 Number bases

In `fmtlib`, you can indicate the base with which to represent an integer by specifying one of `box` for binary, octal, hex respectively.

Code:

```
fmt::print
  ("{0} = {0:b} bin,\n    {0:o} oct,\n
    {0:x} hex\n",
   17);
```

Output

```
[io] fmtbase:
17 = 10001 bin,
    21 oct,
    11 hex
```

7.6.5 Output your own classes

With *fmtlib* this takes a different approach: here you need to specialize the *formatter* struct/class.

Code:

```
1  template <> struct
       fmt::formatter<point> {
2   constexpr
3   auto parse(format_parse_context& ctx)
4         -> decltype(ctx.begin()) {
5     auto it = ctx.begin(),
6       end = ctx.end();
7     if (it != end && *it != '}')
8       throw format_error("invalid
        format");
9     return it;
10   }
11   template <typename FormatContext>
12   auto format
13       (const point& p, FormatContext&
       ctx)
14         -> decltype(ctx.out()) {
15     return format_to
16       (ctx.out(),
17        "{}", p.as_string());
18   }
19 };
20     /* ... */
21     point p(1.1,2.2);
22     fmt::print("{}\n",p);
```

Output
[io] fmtstream:

```
(1.1,2.2)
```

Chapter 8

Lambda expressions

The mechanism of *lambda expression* (first added in C++11 and since expanded) makes dynamic definition of functions possible.

Traditional function usage:
explicitly define a function and apply it:

```
double sum(float x,float y) { return x+y; }
cout << sum( 1.2, 3.4 );
```

New:
apply the function recipe directly:

Code:

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

Output
[func] lambdadirect:

```
3.8
```

This example is of course fairly pointless, but it illustrates the syntax of a lambda expression:

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

- The square brackets, in this case, but not in general, empty, are the *capture* part;
- then follows the usual argument list;
- with a stylized arrow you can indicate the return type, but this is optional if the compiler can figure it out by itself;
- and finally the usual function body, include `return` statement for non-void functions.

Remark 7 *Lambda expressions are sometimes called* closures, *but this term has a technical meaning in programming language theory, that only partly coincides with C++ lambda expressions.*

There are uses for 'Immmediately Invoked Lambda Expression'.

Example: different constructors.

Does not work:

```
if (foo)
  MyClass x(5,5);
else
  MyClass x(6);
```

Solution:

```
MyClass x =
  [foo] () {
    if (foo)
      return MyClass(5,5);
    else
      return MyClass(6);
  }();
```

OpenMP:

```
const int nthreads = [] () -> int {
  int nt;
#pragma omp parallel
#pragma omp master
  nt = omp_get_num_threads();
  return nt;
}();
```

For a slightly more useful example, we can assign the lambda expression to a variable, and repeatedly apply it.

Code:

```
auto summing =
  [] (float x,float y) -> float {
  return x+y; };
cout << summing ( 1.5, 2.3 ) << '\n';
cout << summing ( 3.7, 5.2 ) << '\n';
```

Output
[func] lambdavar:

```
3.8
8.9
```

- This is a variable declaration.
- Uses `auto` for technical reasons; see later.

Return type could have been omitted:

```
auto summing =
[] (float x,float y) { return x+y; };
```

Exercise 8.1. Do exercise 19.10 of the zero finding project.

8.1 Lambda expressions as function argument

Above, when we assigned a lambda expression to a variable, we used `auto` for the type. The reason for this is that each lambda expression gets its own unique type, that is dynamically generated. But that makes it hard to pass that variable to a function.

Suppose we want to pass a lambda expression to a function:

```
int main() {
    apply_to_5( [] (int i) { cout << i+1; } );
```

What type do we use for the function parameter?

```
void apply_to_5( /* what type are we giving? */ f ) {
    f(5);
}
```

Since the type of the lambda expression is dynamically generated, we can not specify that type in the function header.

The way out is to use the *functional* header:

```
#include <functional>
using std::function;
```

With this, you can declare parameters by their signature.

In the following example we write a function *apply_to_5* which

- takes a function *f*, and
- applies it to 5.

We call the *apply_to_5* function with a lambda expression as argument:

Code:

```
1 void apply_to_5
2     ( function< void(int) > f ) {
3   f(5);
4 }
5     /* ... */
6   apply_to_5
7     ( [] (int i) {
8       cout << "Int: " << i << '\n'; } );
```

Output
[func] lambdapass:

Int: 5

Exercise 8.2. Do exercise 19.11 of the zero-finding project.

8.1.1 Lambda members of classes

The fact that a lambda expression has a dynamically generated type also makes it hard to store it in an object. To do this we again use **std**::*function*.

In the following example we make a class *SelectedInts* which takes a boolean function in the constructor: an object will contain only those integers that satisfy the function.

A set of integers, with a test on which ones can be admitted:

```
#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(); };
  std::string string() {
    std::string s;
    for ( int i : bag )
      s += to_string(i)+" ";
    return s;
  };
};
```

We use the above class to construct an object as follows:

- we read an integer `divisor`,
- and accept only those integers into our object that are divisible by that number.

For this we write a lambda expression `is_divisible` that

- captures the divisor, and then
- takes an integer as (its only) argument,
- returning whether that argument is divisible.

Code:

```
1 cout << "Give a divisor: ";
2 cin >> divisor; cout << '\n';
3 cout << ".. using " << divisor
4      << '\n';
5 auto is_divisible =
6   [divisor] (int i) -> bool {
7     return i%divisor==0; };
8 SelectedInts multiples( is_divisible );
9 for (int i=1; i<50; ++i)
10   multiples.add(i);
```

Output
[func] lambdafun:

```
Give a divisor:
.. using 7
Multiples of 7:
7 14 21 28 35 42 49
```

8.2 Captures

A *capture* is a way to 'bake variables into' a function. Let's say we want a function that increments its input, and the increment amount is set when we define the function.

Increment function:

- scalar in, scalar out;
- the increment amount has been fixed through the capture.

Code:

```
1 int one=1;
2 auto increment_by_1 =
3   [one] ( int input ) -> int {
4     return input+one;
5 };
6 cout << increment_by_1 (5)   << '\n';
7 cout << increment_by_1 (12)  << '\n';
8 cout << increment_by_1 (25)  << '\n';
```

Output

```
[func] lambdavalue:

6
13
26
```

Exercise 8.3. Write a program that

- reads a `float` factor;
- defines a function `multiply` that multiplies its input by that factor.

You can capture more than one variable. Explicitly capturing variables is done with a comma-separated list.

Example: multiply by a fraction.

```
int d=2,n=3;
times_fraction = [d,n] (int i) ->int {
    return (i*d)/n;
}
```

Exercise 8.4.

- Set two variables

  ```
  float low = .5, high = 1.5;
  ```

- Define a function of one variable that tests whether that variable is between `low`, `high`. (Hint: what is the signature of that function? What is/are input parameter(s) and what is the return result?)

Exercise 8.5. Do exercises 19.12 and 19.13 of the zero-finding project.

8.2.1 Capture by reference

Normally, captured variables are copied by value.

Attempting to change the captured variable doesn't even compile:

```
auto f = // WRONG DOES NOT COMPILE
    [x] ( float &y ) -> void {
        x *= 2; y += x; };
```

If you do want to alter the captured parameter, pass it by reference:

Code:

```
int stride = 1;
auto more_and_more =
  [&stride] ( int input ) -> void {
    cout << input << "=>" <<
    input+stride << '\n';
    ++stride;
};
more_and_more(5);
more_and_more(6);
more_and_more(7);
more_and_more(8);
more_and_more(9);
cout << "stride is now: " << stride <<
    '\n';
```

Output
[func] lambdareference:

```
5=>6
6=>8
7=>10
8=>12
9=>14
stride is now: 6
```

Capturing by reference can for instance be useful if you are performing some sort of reduction. The capture is then the reduction variable, and the numbers to be reduced come in as function parameter to the lambda expression.

In this example we count how many of the input values test true under a certain function `f`:

Capture a variable by reference so that you can update it:

```
int count=0;
auto count_if_f =
    [&count] (int i) {
      if (f(i)) count++; }
for ( int i : int_data )
  count_if_f(i);
cout << "We counted: " << count;
```

(See the `algorithm` header section 9.3.)

8.2.2 Capturing 'this'

In addition to capturing specific variable, whether by reference or not, as you saw above, you can also capture the whole environment of a lambda. For this the following shorthands exist:

```
[=] () {} // capture everything by value
[&] () {} // capture everything by reference
```

In the context of a class method, this means you are capturing **`this`**. As of C++20, implicit capture of **`this`** by value is deprecated.

8.3 More

8.3.1 Making lambda stateful

Let's consider the issue of lambda expressions and mutable state, by which we mean no more than that a variable gets updated multiple times.

A simple example is a doing a count reduction: how many items satisfy some test. In the following lambda expression, the item is passed as an argument, while the count is captured by reference.

Code:

```
1  vector<int> moreints{8,9,10,11,12};
2  int count{0};
3  for_each
4    ( moreints.begin(),moreints.end(),
5      [&count] (int x) {
6        if (x%2==0)
7          ++count;
8      } );
9  cout << "number of even: " << count
     << '\n';
```

Output
[stl] counteach:

```
number of even: 3
```

How about if that count is not really needed in the calling environment of the lambda expression; can we somehow make it internal?

Lambda expressions are normally stateless:

Code:

```
1 float x = 2, y = 3;
2 auto f = [x] ( float &y ) -> void {
3   int xx = x*2; y += xx; };
4 f(y);
5 cout << y << '\n';
6 f(y);
7 cout << y << '\n';
```

Output
[func] nonmutable:

```
7
11
```

but with the **mutable** keyword you can make them stateful:

Code:

```
1 float x = 2, y = 3;
2 auto f = [x] ( float &y ) mutable ->
    void {
3   x *= 2; y += x; };
4 f(y);
5 cout << y << '\n';
6 f(y);
7 cout << y << '\n';
```

Output
[func] yesmutable:

```
7
15
```

Here is a nifty application: printing a list of numbers, separated by commas, but without trailing comma:

Code:

```
vector x{1,2,3,4,5};
auto printdigit =
  [start=true] (auto xx) mutable ->
  string{
    if (start) {
      start = false;
      return to_string(xx);
    } else
      return ","+to_string(xx);
  };
for ( auto xx : x )
  cout << printdigit(xx);
cout << '\n';
```

Output
[func] lambdaexch:

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

8.3.2 Generic lambdas

The `auto` keyword can be used for *generic lambdas*:

```
auto compare = [] (auto a,auto b) { return a<b; };
```

Here the return type is clear, but the input types are generic. This is much like using a templated function: the compiler instantiates the expression with whatever types are needed.

8.3.3 Algorithms

The `algorithm` header (section 9.3) contains a number of functions that naturally use lambdas. For instance, `any_of` can test whether any element of a `vector` satisfies a condition. You can then use a lambda to specify the `bool` function that tests the condition.

8.3.4 C-style function pointers

The C language had a – somewhat confusing – notation for function pointers. If you need to interface with code that uses them, it is possible to use lambda functions to an extent: lambdas without captures can be converted to a function pointer.

Code:

```
int cfun_add1( int i ) {
  return i+1; };
int apply_to_5( int(*f)(int) ) {
  return f(5); };
//codesnippet  end
   /* ... */
  auto lambda_add1 =
    [] (int i) { return i+1; };
  cout << "C ptr: "
       << apply_to_5(&cfun_add1)
       << '\n';
  cout << "Lambda: "
       << apply_to_5(lambda_add1)
       << '\n';
```

Output
[func] lambdacptr:

```
C ptr: 6
Lambda: 6
```

Chapter 9

Iterators, Algorithms, Ranges

You have seen how you can iterate over a vector

- by an indexed loop over the indices, and
- with a range-based loop over the values.

There is a third way, which is actually the basic mechanism underlying the range-based looping. For this you need to realize that iterating through objects such as vectors isn't simply a process of keeping a counter that says where you are, and taking that element if needed.

Many C++ classes have an *iterator* subclass, that gives a formal description of 'where you are' and what you can find there. Having iterators means that you can traverse structures that don't have an explicit index count, but there are many other conveniences as well.

An *iterator* is, in a metaphorical sense a pointer to a vector element: something that indicates a location in a container such as a vector. Here are some ways iterator are similar to indexes:

- Iteratable containers have a `begin` and `end` iterator.
- The end iterator 'points' just beyond the last element.
- The '`*`' star operator gives the element that the iterator points to.
- You can increment and decrement them (for certain containers).

We start with a discussion of ranges that does not involve iterators, and then we go on to the more general and basic mechanism.

9.1 Ranges

The C++20 standard contains a `ranges` header, which generalizes iteratable objects into as-it-were streams, that can be connected with *pipes*s.

We need to introduced two new concepts.

A `range` is an iteratable object. The containers of the pre-17 STL are ranges, but some new ones have been added.

First of all, ranges provide a clear syntax:

```
vector data{2,3,1};
ranges::sort(data);
```

A `view` is somewhat similar to a range, in the sense that you can iterate over it. The difference is that, unlike for instance a `vector`, a view is not a completely formed object: it is a sort of transformation of a range.

A view doesn't own any data, and any elements you view in it get formed as you iterate over it. This is sometimes called *lazy evaluation* or *lazy execution*. Stated differently, its elements are constructed as they are being requested by the iteration over the view.

In the examples we will write

```
rng::transform
```

and such where

```
namespace rng = std::ranges;
```

However, some features are taken from the `ranges-v3` library, and

```
namespace rng = ranges;
```

9.1.1 Standard algorithms

Many of the STL algorithms now have a range version. Compare two versions of `min_element`, both giving an iterator:

```
vector<float> elements{.5f,1.f,1.5f};
auto min_iter = std::min_element
  (elements.begin(),elements.end());
cout << "Min: " << *min_iter << '\n';
```

```
vector<float> elements{.5f,1.f,1.5f};
auto min_iter =
  rng::min_element(elements);
cout << "Min: " << *min_iter << '\n';
```

9.1.2 Views

A *view* can informally be considered as 'a range that does not own its element'.

Views are composable: you can take one view, and pipe it into another one. If you need the resulting object, rather than the elements as a stream, you can call `to_vector`.

First two simple examples of views:

1. one formed by `transform`, which applies a function to each element of the range or view in sequence;
2. one formed by `filter`, which only yields those elements that satisfy some boolean test.

Code:

```
vector<int> v{ 1,2,3,4,5,6 };
cout << "Original vector: "
     << vector_as_string(v) << '\n';
auto times_two = v
  | rng::views::transform( [] (int i) {
      return 2*i; } );
cout << "Times two: ";
for ( auto c : times_two )
  cout << c << " "; cout << '\n';
auto over_five = times_two
  | rng::views::filter( [] (int i) {
      return i>5; } );
cout << "Over five: ";
for ( auto c : over_five )
  cout << c << " "; cout << '\n';
```

Output
[range] ft1:

```
Original vector: 1, 2, 3, 4,
    5, 6,
Times two: 2, 4, 6, 8, 10,
    12,
Over five: 6, 8, 10, 12,
```

Next to illustrate the composition of streams:

Code:

```
vector<int> v{ 1,2,3,4,5,6 };
cout << "Original vector: "
     << vector_as_string(v) << '\n';
auto times_two_over_five = v
  | rng::views::transform( [] (int i) {
      return 2*i; } )
  | rng::views::filter( [] (int i) {
      return i>5; } );
```

Output
[range] ft2:

```
Original vector: 1, 2, 3, 4,
    5, 6,
Times two over five: 6, 8,
    10, 12,
```

Exercise 9.1. Make a vector that contains both positive and negative numbers. Use ranges to compute the minimum square root of the positive numbers.

Other available operations:

- dropping initial elements: `std::views::drop`
- reversing a vector: `std::views::reverse`

In C++23 some more views have been added.

Zip ranges together with `ranges::views::zip`, giving a tuple:

Code:

```
vector a { 10, 20, 30, 40, 50 };
vector<string> b { "one", "two", "three",
    "four" };

// zip in c++23, not yet in gcc12
for (const auto& [num, name] :
    rng::views::zip(a, b))
  cout << fmt::format("{} -> {}\n", name,
    num);
```

Output
[ranges] zip:

missing snippet
../code/ranges/zip.runout

Return adjacent elements with *ranges::views::adjacent*, giving a tuple.

```
for ( auto [ lt,md,rt ] : values | views::adjacent<3> )
  cout << fixed
       << setw(3) << lt << ","
       << setw(3) << md << ","
       << setw(3) << rt << '\n';
```

Not yet available in my compiler, or in the *range-v3* library.

9.1.3 Example: sum of squares

For computing the sum of squares we can use the range *transform* method for constructing a 'lazy container' of the squares. However, C++20 does not have a range version of the algorithms in *numeric* header, such as *accumulate*. This will be fixed in C++23.

```
vector<float> elements{.5f,1.f,1.5f};
auto squares =
  rng::views::transform(elements, [] (auto e) { return e*e; } );
auto sumsq =
  rng::accumulate( squares, 0.f );
cout << "Sum of squares: " << sumsq << '\n';
```

9.1.4 Infinite sequences

Since views are lazily constructed, it is possible to have an infinite object – as long as you don't ask for its last element.

```
auto result { view::ints(10) // VLE ???
    | views::filter( [] ( const auto& value ) {
                        return value%2==0; } )
    | views::take(10) };
```

9.2 Iterators

Many algorithms that you saw above have an older syntax using iterators.

```
vector data{2,3,1};
// iterator syntax:
sort( begin(data),end(data) );
// range syntax
ranges::sort(data);
```

However, the iterator version is prone to accidents:

```
sort( begin(data1),end(data2) ); // OUCH
```

```
vector<int> myvector(20);
for ( auto copy_of_int : myvector )
  s += copy_of_int;
```

is actually short for:

```
for ( std::vector<int>::iterator
    it=myvector.begin() ;
    it!=myvector.end() ; ++it ) {
      int copy_of_int = *it;
      s += copy_of_int ;
      }
```

Range iterators can be used with anything that is iteratable (vector, map, your own classes!)

There are still some things that you can do with iterators that can not be done with range-based iteration.

Reverse iteration can not be done with range-based syntax.

Use general syntax with reverse iterator: `rbegin`, `rend`.

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

Use `back` to get the value of the last element:

Code:

```
1 vector<int> mydata(5,2);
2 mydata.push_back(35);
3 cout << mydata.size()
4      << '\n';
5 cout << mydata.back()
6      << '\n';
```

Output
[array] vectorend:

```
6
35
```

Set an iterator to the last element and 'dereference' it:

Code:

```
1 vector<int> mydata(5,2);
2 mydata.push_back(35);
3 cout << mydata.size() << '\n';
4 cout << *( --mydata.end() ) << '\n';
```

Output
[array] vectorenditerator:

```
6
35
```

9.2.1 How iterators are like pointers

The container class has a subclass *iterator* that can be used to iterate through all elements of a container.

However, an `iterator` can be used outside of strictly iterating. You can consider an iterator as a sort of 'pointer into a container', and you can move it about.

Let's look at some examples of using the `begin` and `end` iterators. In the following example:

- We first assign the `begin` and `end` iterators to variables; the begin iterator points at the first element, but the end iterator points just beyond the last element;

- Given an iterator, you get the value of the corresponding element by applying the 'star' operator to it;
- A sort of 'pointer arithmetic' can be applied to iterators.

Use independent of looping:

Code:

```
vector<int> v{1,3,5,7};
auto pointer = v.begin();
cout << "we start at "
     << *pointer << '\n';
++pointer;
cout << "after increment: "
     << *pointer << '\n';

pointer = v.end();
cout << "end is not a valid element: "
     << *pointer << '\n';
pointer--;
cout << "last element: "
     << *pointer << '\n';
```

Output

```
[stl] iter:
we start at 1
after increment: 3
end is not a valid element: 0
last element: 7
```

Note that the star notation is a *unary star operator*, not a *pointer dereference*:

```
vector<int> vec{11,22,33,44,55,66};
auto second = vec.begin(); ++second;
cout << "Dereference second: "
     << *second << '\n';
// DOES NOT COMPILE
// the iterator is not a type-star:
// int *subarray = second;
```

9.2.2 Forming sub-arrays

Iterators can be used to construct a `vector`. This can for instance be used to create a *subvector*. In the simplest case, you would make a copy of a vector using begin/end iterators:

```
vector<int> sub( othervec.begin(),othervec.end() );
```

Note that the subvector is formed as a copy of the original elements. Vectors completely 'own' their elements. For non-owning subvectors you would need `span`; section **??**.

Some more examples. We form a subvector:

Code:

```
vector<int> vec{11,22,33,44,55,66};
auto second = vec.begin(); ++second;
auto before = vec.end(); before--;
vector<int>
     sub(vec.data()+1,vec.data()+vec.size()-1);
cout << "no first and last: ";
for ( auto i : sub ) cout << i << ", ";
cout << '\n';
/* ... */
vector<int> vec{11,22,33,44,55,66};
auto second = vec.begin(); ++second;
auto before = vec.end(); before--;
//    vector<int> sub(second,before);
vector<int> sub;
     sub.assign(second,before);
cout << "vector at " <<
     (long)vec.data() << '\n';
cout << "sub at " << (long)sub.data()
     << '\n';

cout << "no first and last: ";
for ( auto i : sub ) cout << i << ", ";
cout << '\n';
vec.at(1) = 222;
cout << "did we get a change in the sub
     vector? " << sub.at(0) << '\n';
```

Output
[iter] subvectorcopy:

```
no first and last: 22, 33,
    44, 55,
```

To demonstrate that the subvector is really a new object, not a subset of the original vector:

Code:

```
    vec.at(1) = 222;
    cout << "did we get a change in the
    sub vector? "
         << sub.at(0) << '\n';
```

Output
[iter] subvectornew:

```
did we get a change in the
    sub vector? 22
```

9.2.3 Vector operations through iterators

You have already seen that the length of a vector can be extended by the `push_back` method (section **??**).

With iterators other operations are possible, such as copying, erasing, and inserting.

First we show the use of `copy` which takes two iterators in one container to define the range to be copied, and one iterator in the target container, which can be the same as the source. The copy operation will overwrite elements in the target, but without bound checking, so make sure there is enough space.

Copy a begin/end range of one container
to an iterator in another container::

Code:

```
vector<int> counts{1,2,3,4};
vector<int> copied(5);
copy( counts.begin(),counts.end(),
      copied.begin()+1 );
cout << copied[0]
     << ", " << copied[1]
     << ".." << copied[4] << '\n';
```

Output
[iter] copy:

```
0, 1..4
```

(No bound checking, so be careful!)

The erase operation `erase` takes two iterators, defining the inclusive lower and exclusive upper bound for the range to erase.

Erase from start to before-end:

Code:

```
vector<int> counts{1,2,3,4,5,6};
vector<int>::iterator second =
    counts.begin()+1;
auto fourth = second+2;
counts.erase(second,fourth);
cout << counts[0]
     << "," << counts[1] << '\n';
```

Output
[iter] erase2:

```
1,4
```

(Also single element without end iterator.)

The `insert` operation takes a target iterator after which the insertion takes place, and two iterators for the range that will be inserted. This will extend the size of the target container.

Insert at iterator: value, single iterator, or range:

Code:

```
vector<int> counts{1,2,3,4,5,6},
  zeros{0,0};
auto after_one = zeros.begin()+1;
zeros.insert
  ( after_one,
    counts.begin()+1,
    counts.begin()+3 );
cout << zeros[0] << ","
     << zeros[1] << ","
     << zeros[2] << ","
     << zeros[3]
     << '\n';
```

Output
[iter] insert2:

```
0,2,3,0
```

9.2.3.1 Indexing and iterating

Functions that would return an array element or location, now return iterators. For instance:

- `find` returns an iterator pointing to the first element equal to the value we are finding;

- `max_element` returns an iterator pointing to the element with maximum value.

One of the arguments for range-based indexing was that we get a simple syntax if we don't need the index. Is it possible to use iterators and still get the index? Yes, that's what the function `distance` is for.

Find 'index' by getting the distance between two iterators:

Code:

```
vector<int> numbers{1,3,5,7,9};
auto it=numbers.begin();
while ( it!=numbers.end() ) {
  auto d = distance(numbers.begin(),it);
  cout << "At distance " << d
       << ": " << *it << '\n';
  ++it;
}
```

Output
[loop] distance:

```
At distance 0: 1
At distance 1: 3
At distance 2: 5
At distance 3: 7
At distance 4: 9
```

Exercise 9.2. Use the above vector methods to return, given a `std::vector<float>`, the integer index of its maximum element.

9.3 Algorithms using iterators

Many simple algorithms on arrays, such as testing 'there is' or 'for all', no longer have to be coded out in C++. They can now be done with a single function from the `std::algorithm` library. This contains 'components that C++ programs may use to perform algorithmic operations on containers and other sequences'.

So, even if you have learned a to code a specific algorithm yourself in the foregoing, you should study the following algorithms, or at least known that such algorithms exist. It's what distinguishes a novice programmer from an industrial-grade (for want of a better term) programmer.

9.3.1 Test Any/all

First we look at some algorithms that apply a predicate to the elements.

- Test if any element satisfies a condition: `any_of`. Note that both `std::any_of` and `std::ranges::any_of` exist. The same holds for the next two.
- Test if all elements satisfy a condition: `all_of`.
- Test if no elements satisfy a condition: `none_of`.
- Apply an operation to all elements: `for_each`.

The object to which the function applies is not specified directly; rather, you have to specify a start and end iterator.

(See chapter 8 for the use of lambda expressions.)

As an example of applying a predicate we look at a couple of examples of using `any_of`. This returns true or false depending on whether the predicate is true for any element of the container; this uses *short-circuit evaluation*.

Reduction with boolean result:
See if any element satisfies a test

Code:

```
#include <ranges>
#include <algorithm>
    /* ... */
    vector<int> ints{1,2,3,4,5,7,8,13,14};
    std::ranges::for_each
      ( ints,
        [] ( int i ) -> void {
          cout << i << '\n';
        }
        );
```

Output
[iter] eachr:

```
1
2
3
4
5
7
8
13
14
```

(Why wouldn't you use a `accumulate` reduction?)

Here is an example using `any_of` to find whether a certain element appears in a vector:

Code:

```
vector<int> ints{1,2,3,4,5,7,8,13,14};
bool there_was_an_8 =
  std::ranges::any_of
  ( ints,
    [] ( int i ) -> bool {
      return i==8;
    }
    );
cout << "There was an 8: " << boolalpha
     << there_was_an_8 << '\n';
```

Output
[iter] anyr:

```
There was an 8: true
```

Capturing the value to be tested gives:

Code:

```
vector<int> ints{1,2,3,4,5,7,8,13,14};
int tofind = 8;
bool there_was_an_8 =
  std::ranges::any_of
  ( ints,
    [tofind] ( int i ) -> bool {
      return i==tofind;
    }
    );
cout << "There was an 8: " << boolalpha
     << there_was_an_8 << '\n';
```

Output
[iter] anyc:

```
There was an 8: true
```

Remark 8 *The previous examples rely on C++20 ranges. With iterators they look like this:*

Code:

```
    vector<int>
    ints{2,3,4,5,7,8,13,14,15};
    bool there_was_an_8 =
      any_of( ints.begin(),ints.end(),
              [] ( int i ) -> bool {
                return i==8;
              }
              );
    cout << "There was an 8: " <<
    boolalpha << there_was_an_8 << '\n';
```

Output
[iter] any:

```
There was an 8: true
```

Code:

```
#include <algorithm>
    /* ... */
    vector<int>
    ints{2,3,4,5,7,8,13,14,15};
    for_each( ints.begin(),ints.end(),
              [] ( int i ) -> void {
                cout << i << '\n';
              }
              );
```

Output
[iter] each:

```
2
3
4
5
7
8
13
14
15
```

9.3.2 Apply to each

The `for_each` algorithm applies a function to every element of a container. Unlike the previous algorithms, this can alter the elements.

To introduce the syntax, we look at the pointless example of outputting each element:

Code:

```
#include <algorithm>
using std::for_each;
    /* ... */
  vector<int> ints{3,4,5,6,7};
  for_each
    ( ints.begin(),ints.end(),
      [] (int x) -> void {
        if (x%2==0)
          cout << x << '\n';
      } );
```

Output
[stl] printeach:

```
4
6
```

Apply something to each array element:

Code:

```
#include <ranges>
#include <algorithm>
    /* ... */
    vector<int> ints{1,2,3,4,5,7,8,13,14};
    std::ranges::for_each
      ( ints,
        [] ( int i ) -> void {
          cout << i << '\n';
        }
        );
```

Output
[iter] eachr:

```
1
2
3
4
5
7
8
13
14
```

Exercise 9.3. Use *for_each* to sum the elements of a vector.

Hint: the problem is how to treat the sum variable. Do not use a global variable!

Capture by reference, to update with the array elements.

Code:

```
    vector<int>
    ints{2,3,4,5,7,8,13,14,15};
    int sum=0;
    for_each( ints.begin(),ints.end(),
              [&sum] ( int i ) -> void {
                sum += i;
              }
              );
    cout << "Sum = " << sum << '\n';
```

Output
[iter] each:

```
2
3
4
5
7
8
13
14
15
```

9.3.3 Iterator result

Some algorithms do not result in a value, but rather in an iterator that points to the location of that value. Examples: *min_element* takes a begin and end iterator, and returns the iterator in between where the minimum element is found. To find the actual value, we need to 'dereference' the iterator:

```
vector<float> elements{.5f,1.f,1.5f};
auto min_iter = std::min_element
  (elements.begin(),elements.end());
cout << "Min: " << *min_iter << '\n';
```

Similarly *max_element*.

9.3.4 Mapping

The *transform* algorithm applies a function to each container element, modifying it in place:

```
std::transform( vec, vec.begin(), [] (int i) { return i*i; } );
```

9.3.5 Reduction

Numerical *reductions* can be applied using iterators in `accumulate` in the `numeric` header. If no reduction operator is specified, a *sum reduction* is performed.

Default is sum reduction:

Code:

```
#include <numeric>
using std::accumulate;
    /* ... */
    vector<int> v{1,3,5,7};
    auto first = v.begin();
    auto last  = v.end();
    auto sum = accumulate(first,last,0);
    cout << "sum: " << sum << '\n';
```

Output

```
[stl] accumulate:
sum: 16
```

Other binary *arithmetic operators* that can be used as *reduction operator* are found in `functional`:

- `plus`, `minus`, `multiplies`, `divides`,
- integers only: `modulus`
- boolean: `logical_and`, `logical_or`

This header also contains the unary `negate` operator, which can of course not be used for reductions.

As an example of an explicitly specified reduction operator:

```
auto p = std::accumulate
  ( x.begin(),x_end(),1.f,
    std::multiplies<float>()
  );
```

Note:

- that the operator is templated, and that it is followed by parentheses to become a functor, rather than a class;
- that the accumulate function is templated, and it takes its type from the init value. Thus, in the above example, a value of `1` would have turned this into an integer operation.

Using lambda functions (chapter 8) we can get more complicated effects.

Supply multiply operator:

Code:

```
using std::multiplies;
    /* ... */
    vector<int> v{1,3,5,7};
    auto first = v.begin();
    auto last  = v.end();
    ++first; last--;
    auto product =
      accumulate(first,last,2,
                 multiplies<>());
    cout << "product: " << product <<
    '\n';
```

Output

[stl] product:

```
product: 30
```

Specific for the max reduction is `max_element`. This can be called without a comparator (for numerical max), or with a comparator for general maximum operations. The maximum and minimum algorithms return an iterator, rather than only the max/min value.

Example: maximum relative deviation from a quantity:

```
max_element(myvalue.begin(),myvalue.end(),
    [my_sum_of_squares] (double x,double y) -> bool {
        return fabs( (my_sum_of_squares-x)/x ) < fabs( (my_sum_of_squares-y)/y
    ); }
);
```

For more complicated lambdas used in `accumulate`,

- the first argument should be the reduce type,
- the second argument should be the iterated type

In the following example we accumulate one member of a class:

```
class x {
public:
  int i,j;
  x() {};
  x(int i,int j) : i(i),j(j) {};
};
```

```
std::vector< x > xs(5);
auto xxx =
  std::accumulate
  ( xs.begin(),xs.end(),0,
    [] ( int init,x x1 ) -> int
{ return x1.i+init; }
    );
```

9.3.6 Sorting

The `algorithm` header also has a function `sort`.

With iterators you can easily apply this to things such as vectors:

```
sort( myvec.begin(),myvec.end() );
```

The comparison used by default is ascending. You can specify other compare functions:

```
sort( myvec.begin(),myvec.end(),
      [] (int i,int) { return i>j; }
    );
```

or

```
sort( people.begin(),people.end(),
      [] ( const Person& lhs,const Person& rhs ) {
         return lhs.name < rhs.name; }
    )
```

With iterators you can also do things like sorting a part of the vector:

Code:

```
1 vector<int> v{3,1,2,4,5,7,9,11,12,8,10};
2 cout << "Original vector: " <<
      vector_as_string(v) << '\n';
3
4 auto v_std(v);
5 std::sort(
      v_std.begin(),v_std.begin()+5 );
6 cout << "Five elements sorts: " <<
      vector_as_string(v_std) << '\n';
```

Output
[range] sortit:

```
Original vector: 3, 1, 2, 4,
    5, 7, 9, 11, 12, 8, 10,
Five elements sorts: 1, 2,
    3, 4, 5, 7, 9, 11, 12, 8,
    10,
```

9.3.7 Parallel execution policies

The C++17 standard added the `ExecutionPolicy` concept to standard algorithms, describing how an element of the `algorithm` library may be executed in parallel.

There are three choices for the *execution policy*, defined in the `execution` header:

- `std::execution::seq`: iterations may not be parallelized, but are indeterminedly sequenced in the evaluation thread.
- `std::execution::par`: iterations may be executed in parallel, but are indetermined sequences;
- `std::execution::par_unseq`: iterations may be parallelized, vectorized, migrated over threads.
- `std::execution::unseq` (since C++20): iterations are allowed to be vectorized over.

To accomodate the indeterminate evaluation order, new 'unordered' algorithms are introduced based on existing 'ordered algorithms:

- `reduce`: similar `accumulate`, but unordered and therefore parallelizable through an `ExecutionPolicy`;
- `inclusive_scan`: similar to `partial_sum`;
- `exclusive_scan`: no ordered equivalent
- `transform_reduce`, `transform_inclusive_scan`, `transform_exclusive_scan`.

```
transform_reduce( execpol, first,last, init, reduce_op,transform_op );
```

Some performance measurements on these are given in MPI/OpenMP book [10], section 19.4.

9.3.8 Classification of algorithms

(Taken from a lecture by Dietmar Kühl at CppCon 2017, as per its copyright notice.)

9.3.8.1 Non-parallel algorithms

9.3.8.1.1 **Order $O(1)$ algorithms** `clamp`, `destroy_at`, `gcd`, `iter_swap`, `lcm`, `max`, `min`, `minmax`,

9.3.8.1.2 **Order $O(\log n)$ algorithms** `binary_search`, `equal_range`, `lower_bound`, `partition_point`, `pop_heap`, `push_heap`, `upperbound`,

9.3.8.1.3 **Heap algorithms** `make_heap`, `sort_heap`,

9.3.8.1.4 **Permutation algorithms** `is_permutation`, `next_permutation`, `prev_permutation`,

9.3.8.1.5 **Overlapping algorithms** `copy_backward`, `move_backward`,

9.3.8.1.6 **Renamed algorithms** `accumulate`, `partial_sum`,

9.3.8.1.7 **Oddball algorithms** `iota`, `sample`, `shuffle`,

9.3.8.2 Parallel algorithms

9.3.8.2.1 **Map algorithms** `copy`, `copy_n`, `destroy`, `destroy_n`, `fill`, `fill_n`, `for_each`, `for_each_n`, `generate`, `generate_n`, `move`, `replace`, `replace_copy`, `replace_copy_if`, `replace_if`, `reverse`, `reverse_copy`, `swap_ranges`, `transform`, `uninitalized_*`,

9.3.8.2.2 **Reduce algorithms** `adjacent_find`, `all_of`, `any_of`, `count`, `count_if`, `equal`, `find`, `find_end`, `find_first_of`, `find_if`, `find_if_not`, `includes`, `inner_product`, `is_heap`, `is_heap_until`, `is_partitioned`, `is_sorted`, `is_sorted_until`, `lexicographical_compare`, `max_element`, `min_element`, `minmax_element`, `mismatch`, `none_of`, `reduce`, `search`, `search_n`,

9.3.8.2.3 **Scan algorithms** `exclusive_scan`, `inclusive_scan`,

9.3.8.2.4 **Fused algorithms** `transform_exclusive_scan`, `transform_inclusive_scan`, `transform_reduce`,

9.3.8.2.5 **Gather algorithms** `copy_if`, `partition_copy`, `remove`, `remove_copy`, `remove_copy_if`, `remove_if`, `rotate`, `unique`, `unique_copy`,

9.3.8.2.6 **Special algorithms** *adjacent_difference*, *inplace_merge*, *merge*, *nth_element*, *partial_sort*, *partial_sort_copy*, *partition*, *rotate*, *set_difference*, *set_intersection*, *set_symmetric_difference*, *set_union*, *sort*, *stable_partition*, *stable_sort*,

9.4 Advanced topics

9.4.1 Range types

Types of ranges:

- **std**::*ranges*::*input_range* : iterate forward at least once, as if you're accepting input with **cin** and such.
- **std**::*ranges*::*forward_range* : can be iterated forward, (for instance with plus-plus), multiple times, as in a *single-linked list*.
- **std**::*ranges*::*bidirectional_range* : can be iterated in both directions, for instance with plus-plus and minus-minus.
- **std**::*ranges*::*random_access_range* items can be found in constant time, such as with square bracket indexing.
- **std**::*ranges*::*contiguous_range* : items are stored consecutively in memory, making address calculations possible.

9.4.2 Make your own iterator

You know that you can iterate over `vector` objects:

```
vector<int> myvector(20);
for ( auto copy_of_int : myvector )
  s += copy_of_int;
for ( auto &ref_to_int : myvector )
  ref_to_int = s;
```

(Many other STL classes are iteratable like this.)

This is not magic: it is possible to iterate over any class: a *class* is *iteratable* that has a number of conditions satisfied.

The class needs to have:

- a method *begin* with prototype

  ```
  iteratableClass iteratableClass::begin()
  ```

  That gives an object in the initial state, which we will call the 'iterator object'; likewise
- a method *end*

  ```
  iteratableClass iteratableClass::end()}
  ```

  that gives an object in the final state; furthermore you need
- an increment operator

  ```
  void iteratableClass::operator++()
  ```

  that advances the iterator object to the next state;
- a test

```
bool iteratableClass::operator!=(const iteratableClass&)
```

to determine whether the iteration can continue; finally

- a dereference operator

```
iteratableClass::operator*()
```

that takes the iterator object and returns its state.

All this was visible in pre-C++11 iterating, where a loop over a vector looked like:

```
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++. However, you're applying it to an iterator, not a pointer, and this is an operator you are applying.
- 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.

9.4.2.1 Example 1

Let's make a class, called a `bag`, that models a set of integers, and we want to enumerate them. For simplicity sake we will make a set of contiguous integers:

```
class bag {
  // basic data
private:
  int first,last;
public:
  bag(int first,int last) : first(first),last(last) {};
```

When you create an iterator object it will be copy of the object you are iterating over, except that it remembers how far it has searched:

```
class iter {
private:
  int seek{0};
public:
  iter(int i) : seek(i) {};
  int value() { return seek; };
```

The `begin` method gives a `bag` with the seek parameter initialized:

```
public:
  iter begin() { return iter(first); };
  iter end()   { return iter(last); };
```

These routines are public because they are (implicitly) called by the client code.

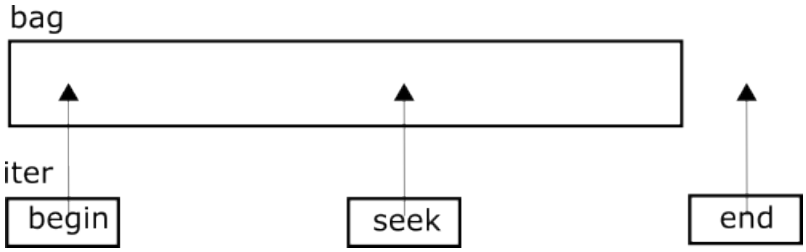


Figure 9.1: Iterator objects into the `bag` object

The termination test method is called on the iterator, comparing it to the `end` object:

```
bool operator!=( const iter &test ) const { return seek!=test.seek; };
bool operator==( const iter &test ) const { return seek==test.seek; };
```

Finally, we need the increment method and the dereference. Both access the `seek` member:

```
void operator++() { ++seek; };
int operator*() { return seek; };
```

We can iterate over our own class:

Code:

```
1  bag digits(0,9);
2
3  bool find3{false};
4  for ( auto seek : digits )
5    find3 = find3 || (seek==3);
6  cout << "found 3: " << boolalpha
7       << find3 << '\n';
8
9  bool find15{false};
10 for ( auto seek : digits )
11   find15 = find15 || (seek==15);
12 cout << "found 15: " << boolalpha
13      << find15 << '\n';
```

Output

```
[loop] bagfind:

found 3: true
found 15: false
```

(for this particular case, use `std::any_of`)

Code:

```
// DOES NOT WORK YET: NEEDS
     iterator_category
// bool find8 = any_of
//    ( digits.begin(),digits.end(),
//      [=] (bag::iter i) { return
     i.value()==8; } );
// cout << "found 8: " << boolalpha
//      << find8 << '\n';
```

Output

```
[loop] bagany:
found 8: true
```

If we add a method `has` to the class:

```
bool has(int tst) {
  for (auto seek : *this )
    if (seek==tst) return true;
  return false;
};
```

we can call this:

```
cout << "f3: " << digits.has(3) << '\n';
cout << "f15: " << digits.has(15) << '\n';
```

Of course, we could have written this function without the range-based iteration, but this implementation is particularly elegant.

Exercise 9.4. You can now do exercise 17.24, implementing a prime number generator with this mechanism.

If you think you understand `const`, consider that the `has` method is conceptually `cost`. But if you add that keyword, the compiler will complain about that use of `*this`, since it is altered through the `begin` method.

Exercise 9.5. Find a way to make `has` a `const` method.

9.4.2.2 *Example 2: iterator class*

Recall that

Short hand:

```
vector<float> v;
for ( auto e : v )
  ... e ...
```

for:

```
for ( vector<float>::iterator
    e=v.begin();
      e!=v.end(); e++ )
  ... *e ...
```

If we want

```
for ( auto e : my_object )
```

```
... e ...
```

we need an iterator class with methods such as `begin`, `end`, `*` and `++`.

Ranging over a class with iterator subclass

Class:

```
class NewVector {
protected:
  // vector data
  int *storage;
  int s;
    /* ... */
public:
  // iterator stuff
  class iter;
  iter begin();
  iter end();

};
```

Main:

```
NewVector v(s);
  /* ... */
for ( auto e : v )
  cout << e << " ";
```

Random-access iterator:

```
NewVector::iter& operator++();
int& operator*();
bool operator==( const NewVector::iter &other ) const;
bool operator!=( const NewVector::iter &other ) const;
// needed to OpenMP
int operator-( const NewVector::iter& other ) const;
NewVector::iter& operator+=( int add );
```

Exercise 9.6. Write the missing iterator methods. Here's something to get you started.

```
class NewVector::iter {
private: int *searcher;
    /* ... */
NewVector::iter::iter( int *searcher )
  : searcher(searcher) {};
NewVector::iter NewVector::begin() {
  return NewVector::iter(storage); };
NewVector::iter NewVector::end()   {
  return NewVector::iter(storage+NewVector::s); };
```

Chapter 10

References

10.1 Reference

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

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

Code:

```
1 // /localparm.cpp
2 void change_scalar(int i) {
3   i += 1;
4 }
5     /* ... */
6 // /localparm.cpp
7   number = 3;
8   cout << "Number is 3: "
9        << number << '\n';
10  change_scalar(number);
11  cout << "is it still 3? Let's see: "
12       << number << '\n';
```

Output
[func] localparm:

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

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

```
// /arraypass.cpp
void change_scalar_by_reference(int &i) { i += 1; }
```

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

10.2 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.

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
```

```
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 ) {
    .... };
```

10.3 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; };
};
```

Now the return statement makes a copy of `thing`, which may be the desired behavior, and it may not be. If you don't need an actual copy, but you want access to the actual data member, you can return the member by *reference*:

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

Now you have write access to an internal (maybe private!) data member. You may want to have that, but if you don't use a *const reference*:

Code:

```
1 class has_int {
2 private:
3   int mine{1};
4 public:
5   const int& int_to_get() { return
      mine; };
6   int& int_to_set() { return mine; };
7   void inc() { ++mine; };
8 };
9     /* ... */
10  has_int an_int;
11  an_int.inc(); an_int.inc();
      an_int.inc();
12  cout << "Contained int is now: "
13       << an_int.int_to_get() << '\n';
14  /* Compiler error:
      an_int.int_to_get() = 5; */
15  an_int.int_to_set() = 17;
16  cout << "Contained int is now: "
17       << an_int.int_to_get() << '\n';
```

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:

```
1  myclass obj(5);
2  cout << "object data: "
3       << obj.data() << '\n';
4  int dcopy = obj.data();
5  ++dcopy;
6  cout << "object data: "
7       << obj.data() << '\n';
8  int &dref = obj.data();
9  ++dref;
10 cout << "object data: "
11      << obj.data() << '\n';
12 auto dauto = obj.data();
13 ++dauto;
14 cout << "object data: "
15      << obj.data() << '\n';
16 auto &aref = obj.data();
17 ++aref;
18 cout << "object data: "
19      << obj.data() << '\n';
```

Output
[func] rhsref:

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

(On the other hand, after `const auto &ref` the reference is not modifiable. This variant is useful if you want read-only access, without the cost of copying.)

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.

10.4 Reference to array members

You can define various operator, such as `+-*/` arithmetic operators, to act on classes, with your own provided implementation; see section **??**. 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 access 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:

```
// /getindex1.cpp
class vector10 {
private:
  int array[10];
public:
    /* ... */
// /getindex1.cpp
  int operator()(int i) {
    return array[i];
  };
  int operator[](int i) {
```

```
    return array[i];
  };
};
    /* ... */
// /getindex1.cpp
  vector10 v;
  cout << v(3) << '\n';
  cout << v[2] << '\n';
  /* 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:

```
// /getindex2.cpp
  int& operator[](int i) {
    return array[i];
  };
};
    /* ... */
// /getindex2.cpp
  cout << v[2] << '\n';
  v[2] = -2;
  cout << v[2] << '\n';
```

Your reason for wanting to return a reference could be 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*:

```
// /getindex3.cpp
  const int& operator[](int i) {
    return array[i];
  };
};
    /* ... */
// /getindex3.cpp
  cout << v[2] << '\n';
  /* compilation error: v[2] = -2; */
```

10.5 rvalue references

See the chapter about obscure stuff; section 15.4.3.

Chapter 11

Pointers

Pointers are an indirect way of associating an entity with a variable name. Consider for instance the circular-sounding definition of a list:

> A list consists of nodes, where a node contains some information in its 'head', and which has a 'tail' that is a list.

Naive code:

```
class Node {
private:
  int value;
  Node tail;
  /* ... */
};
```

This does not work: would take infinite memory.

Indirect inclusion: only 'point' to the tail:

```
class Node {
private:
  int value;
  PointToNode tail;
  /* ... */
};
```

This chapter will explain C++ *smart pointer*s, and give some uses for them.

11.1 Pointer usage

We will focus on pointers to objects of some class.

Simple class that stores one number:

```
class HasX {
private:
  double x;
public:
  HasX( double x) : x(x) {};
```

```
    auto get() { return x; };
    void set(double xx) { x = xx; };
  };
```

With class objects, the 'dot' notation for class members becomes an 'arrow' notation when you use a pointer. 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`.

- If `x` is object with member `y`:
  `x.y`
- If `xx` is pointer to object with member `y`:
  `xx->y`
- Arrow notation works with old-style pointers and new shared/unique pointers.

Instead of creating an object, you now create an object with a pointer to it, in a single call.

Note that you don't first create an object, and then set a pointer to it, the way it happens in many other languages. Smart pointers work differently: you create the object and the pointer to it in one call.

```
make_shared< ClassName >( constructor arguments );
```

The resulting object is of type `shared_ptr<ClassName>`, but you can save yourself spelling that out, and use `auto` instead.

Object vs pointed-object:

Code:

```
1  #include <memory>
2  using std::make_shared;
3
4     /* ... */
5    HasX xobj(5);
6    cout << xobj.value() << '\n';
7    xobj.set(6);
8    cout << xobj.value() << '\n';
9
10   auto xptr = make_shared<HasX>(5);
11   cout << xptr->value() << '\n';
12   xptr->set(6);
13   cout << xptr->value() << '\n';
```

Output

```
[pointer] pointx:

5
6
5
6
```

Using smart pointers requires at the top of your file:

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

using std::unique_ptr;
using std::make_unique;
```

Why do we use pointers?
Pointers make it possible for two variables to own the same object.

Code:

```
1 auto xptr = make_shared<HasX>(5);
2 auto yptr = xptr;
3 cout << xptr->get() << '\n';
4 yptr->set(6);
5 cout << xptr->get() << '\n';
```

Output
[pointer] twopoint:

```
5
6
```

The prototypical example for the use of pointers is in linked lists and graph data structures. See section **??** for code and discussion.

Exercise 11.1. If you are doing the geometry project (chapter 18) you can now do exercise 18.16.

11.2 Memory leaks and 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.

We need a class with constructor and destructor tracing:

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

Just to illustrate that the destructor gets called when the object goes out of scope:

Code:

```
1 cout << "Outside\n";
2 {
3   thing x;
4   cout << "create done\n";
5 }
6 cout << "back outside\n";
```

Output
[pointer] ptr0:

```
Outside
.. calling constructor
create done
.. calling destructor
back outside
```

Now illustrate that the destructor of the object is called when the pointer no longer points to the object. We do this by assigning `nullptr` to the pointer. (This is very different from `NULL` in C: the null pointer is actually an object with a type; see section 11.3.7.)

Let's create a pointer and overwrite it:

Code:

```
1 cout << "set pointer1"
2      << '\n';
3 auto thing_ptr1 =
4   make_shared<thing>();
5 cout << "overwrite pointer"
6      << '\n';
7 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.

Code:

```
1 cout << "set pointer2" << '\n';
2 auto thing_ptr2 =
3   make_shared<thing>();
4 cout << "set pointer3 by copy"
5      << '\n';
6 auto thing_ptr3 = thing_ptr2;
7 cout << "overwrite pointer2"
8      << '\n';
9 thing_ptr2 = nullptr;
10 cout << "overwrite pointer3"
11      << '\n';
12 thing_ptr3 = nullptr;
```

Output
[pointer] ptr2:

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

- The object counts how many pointers there are:
- 'reference counting'
- A pointed-to object is deallocated if no one points to it.

Remark 9 *A more obscure source of memory leaks has to do with exceptions:*

```
void f() {
  double *x = new double[50];
  throw("something");
```

```
    delete x;
}
```

Because of the exception (which can of course come from a nested function call) the `delete` *statement is never reached, and the allocated memory is leaked. Smart pointers solve this problem.*

11.3 Advanced topics

11.3.1 Pointers and arrays

The constructor syntax is a little involved for vectors:

```
auto x = make_shared<vector<double>>(vector<double>{1.1,2.2});
```

11.3.2 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.

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

Code:

```
1 auto Y = make_shared<HasY>(5);
2 cout << Y->y << '\n';
3 Y.get()->y = 6;
4 cout << ( *Y.get() ).y << '\n';
```

Output

[pointer] pointy:

```
5
6
```

11.3.3 Unique pointers

Shared pointers are fairly easy to program, and they come with lots of advantages, such as the automatic memory management. However, they have more overhead than strictly necessary because they have a *reference count* mechanism to support the memory management. Therefore, there exists a *unique pointer*, `unique_ptr`, for cases where an object will only ever be 'owned' by one pointer. In that case, you can use a C-style *bare pointer* for non-owning references.

11.3.4 Base and derived pointers

Suppose you have base and derived classes:

```
class A {};
class B : public A {};
```

Just like you could assign a `B` object to an `A` variable:

```
B b_object;
A a_object = b_object;
```

is it possible to assign a `B` pointer to an `A` pointer?

The following construct makes this possible:

```
auto a_ptr = shared_pointer<A>( make_shared<B>() );
```

Note: this is better than

```
auto a_ptr = shared_pointer<A>( new B() );
```

Again a reason we don't need **new** anymore!

11.3.5 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>
```

Note that you can only return a `shared_from_this` if already a valid shared pointer to that object exists.

11.3.6 Weak pointers

In addition to shared and unique pointers, which own an object, there is also `weak_ptr` which creates a *weak pointer*. This pointer type does not own, but at least it knows when it dangles.

```
weak_ptr<R> wp;
shared_ptr<R> sp( new R );
sp->call_member();
wp = sp;
// access through new shared pointer:
auto p = wp.lock();
```

```
if (p) {
  p->call_member();
}
if (!wp.expired()) { // not thread-safe!
  wp.lock()->call_member();
}
```

There is a subtlety with weak pointers and shared pointers. The call

```
auto sp = shared_ptr<Obj>( new Obj );
```

creates first the object, then the 'control block' that counts owners. On the other hand,

```
auto sp = make_shared<Obj>();
```

does a single allocation for object and control block. However, if you have a weak pointer to such a single object, it is possible that the object is destructed, but not de-allocated. On the other hand, using

```
auto sp  = shared_ptr<Obj>( new Obj );
```

creates the control block separately, so the pointed object can be destructed and de-allocated. Having a weak pointer to it means that only the control block needs to stick around.

11.3.7 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.

Unfortunately, *dereferencing a nullptr* does not give an exception.

11.3.8 Opaque pointer

The need for *opaque pointers* `void*` is a lot less in C++ than it was in C. For instance, contexts can often be modeled with captures in closures (chapter 8). If you really need a pointer that does not *a priori* knows what it points to, use `std::any`, which is usually smart enough to call destructors when needed. See section **??** for details.

Code:

```
std::any a = 1;
cout << a.type().name() << ": "
     << std::any_cast<int>(a) << '\n';
a = 3.14;
cout << a.type().name() << ": "
     << std::any_cast<double>(a) <<
    '\n';
a = true;
cout << a.type().name() << ": "
     << std::any_cast<bool>(a) << '\n';

try {
  a = 1;
  cout << std::any_cast<float>(a) <<
    '\n';
} catch (const std::bad_any_cast& e) {
  cout << e.what() << '\n';
}
```

Output
[pointer] any:

```
i: 1
d: 3.14
b: true
bad any cast
```

11.3.9 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.

Here is an example of a pointer to a solitary double:

Code:

```
// shared pointer to allocated double
auto array = shared_ptr<double>( new
    double );
double *ptr = array.get();
array.get()[0] = 2.;
cout << ptr[0] << '\n';
```

Output
[pointer] ptrdouble:

```
2
```

It is possible to initialize that double:

Code:

```
// shared pointer to initialized double
auto array = make_shared<double>(50);
double *ptr = array.get();
cout << ptr[0] << '\n';
```

Output
[pointer] ptrdoubleinit:

```
50
```

11.4 Smart pointers vs C pointers

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 chapter **??**.
- Arrays can largely be done through `std::vector`, rather than `malloc`; see chapter **??**.
- Traversing arrays and vectors can be done with ranges; section **??**.
- Anything that obeys a scope should be created through a *constructor*, rather than using `malloc`.

11.4.1 Smart pointers versus C-style address pointers

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

```
auto
  p = shared_ptr<HasY>( &y );
p->y = 3;
cout << "Pointer's y: "
     << p->y << '\n';
```

gives:

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

Smart pointers are much better than old style pointers

```
Obj *X;
*X = Obj( /* args */ );
```

There is a final way of creating a shared pointer where you cast an old-style `new` object to shared pointer

```
auto p = shared_ptr<Obj>( new Obj );
```

This is not the preferred mode of creation, but it can be useful in the case of *weak pointers*; section 11.3.6.

Chapter 12

Templates

You have seen in this course how objects of type `vector<string>` and `vector<float>` are very similar: have methods with the same names, and these methods behave largely the same. This angle-bracket notation is called 'templating' and the `string` or `float` is called the *template parameter*.

If you go digging into the source of the C++ library, you will find that, somewhere, there is just a single definition of the `vector` class, but with a new notation it gets the type `string` or `float` as template parameter.

To be precise, the templated class (or function) is preceded by a line

```
template< typename T >
class vector {
  ...
};
```

This mechanism is available for your own functions and classes too.

If you have multiple routines that do 'the same' for multiple types, you want the type name to be a variable, a *template parameter*. Syntax:

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

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

12.1 Templated functions

We will start by taking a brief look at templated functions.

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 12.1. 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 << "Epsilon float: "
     << setw(10) << setprecision(4)
     << float_eps << '\n';

double double_eps;
epsilon(double_eps);
cout << "Epsilon double: "
     << setw(10) << setprecision(4)
     << double_eps << '\n';
```

Output

```
[template] eps:

Epsilon float: 1.0000e-07
Epsilon double: 1.0000e-15
```

Exercise 12.2. If you're doing the zero-finding project, you can now do the exercises in section 19.2.3.

12.2 Templated classes

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

The STL contains in effect

```
template<typename T>
class vector {
private:
  T *vectordata; // internal data
public:
  T at(int i) { return vectordata[i]  };
  int size() { /* return size of data */ };
  // much more
}
```

Let's consider a worked out example. We write a class *Store* that stored a single element of the type of the template parameter:

Code:

```
Store<int> i5(5);
cout << i5.value() << '\n';
```

Output

```
[template] example1i5:

5
```

The class definition looks pretty normal, except that the type (**int** in the above example) is parametrized:

```
template< typename T >
class Store {
private:
  T stored;
public:
  Store(T v) : stored(v) {};
  T value() { return stored;};
```

If we write methods that refer the templated type, things get a little more complicated. Let's say we want two methods `copy` and `negative` that return objects of the same templated type:

Code:

```
1 Store<float> also314 = f314.copy();
2 cout << also314.value() << '\n';
3 Store<float> min314 = f314.negative();
4 cout << min314.value() << '\n';
```

Output
[template] example1f314:

```
3.14
-3.14
```

The method definitions are fairly straightforward; if you leave out the template parameter, the *class name injection* mechanism uses the same template value as for the class being defined:

```
Store copy() { return Store(stored); };
Store<T> negative() { return Store<T>(-stored); };
```

12.2.1 Out-of-class method definitions

If we separate the class signature and the method definitions (for instance for separate compilation; see section **??**) things get trickier. The class signature is easy:

```
template< typename T >
class Store {
private:
  T stored;
public:
  Store(T v);
  T value();
  Store copy();
  Store<T> negative();
};
```

The method definitions are more tricky. Now the template parameter needs to be specified every single time you mention the templated class, except for the name of the constructor:

```
template< typename T >
Store<T>::Store(T v) : stored(v) {};

template< typename T >
T Store<T>::value() { return stored;};

template< typename T >
Store<T>  Store<T>::copy() { return Store<T>(stored); };

template< typename T >
Store<T> Store<T>::negative() { return Store<T>(-stored); };
```

12.2.2 Specific implementations

Sometimes the template code works for a number of types (or values), but not for all. In that case you can specify the instantiation for specific types with empty angle brackets:

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

12.2.3 Templates and separate compilation

The use of templates often makes *separate compilation* hard: in order to compile the templated definitions the compiler needs to know with what types they will be used. For this reason, many libraries are *header-only*: they have to be included in each file where they are used, rather than compiled separately and linked.

In the common case where you can foresee with which types a templated class will be instantiated, there is a way out. Suppose you have a templated class and function:

```
template <typename T>
class foo<T> {
};

template<typename T>
double f( T x );
```

and they will only be used (instantiated) with `float` and `double`, then adding the following lines after the class definition makes the file separately compilable:

```
template class foo<float>;
template class foo<double>;

template double f(float);
template double f(double);
```

If the class is split into a header and implementation file, these lines go in the implementation file.

12.3 Example: polynomials over fields

Any numerical application can be templated to allow for computation in *single precision* `float`s, and *double precision* `double`s. However, we can often also generalize the computation to other *fields*. Consider as an example polynomials, in both scalars and (square) matrices.

Let's start with a simple class for polynomials:

```
class polynomial {
private:
  vector<double> coefficients;
public:
  polynomial( vector<double> c )
```

```
    : coefficients(c) {};
  // 5 x^2 + 4 x + 3 = 5 x + 4 x + 3
  double eval( double x ) const {
    double y{0.};
    for_each(coefficients.rbegin(),coefficients.rend(),
             [x,&y] (double c) { y *= x; y += c; } );
    return y;
  };
  double operator()(double x) const { return eval(x); };
```

We store the polynomial coefficients, with the zeroth-degree coefficient in location zero, et cetera. The routine for evaluating a polynomial for a given input x is an implementation of *Horner's scheme*:

$$5x^2 + 4x + 3 \equiv ((5) \cdot x + 4) \cdot x + 3$$

(Note that the `eval` method above uses `rbegin`, `rend` to traverse the coefficients in the correct order.)

For instance, the coefficients `2,0,1` correspond to the polynomial $2 + 0 \cdot x + 1 \cdot x^2$:

```
polynomial x2p2( {2., 0., 1.} );
for ( auto x : {1., 2., 3.} ) {
  auto y = x2p2(x);
  cout << "Second power of x=" << x << " plus 2 gives y=" << y << '\n';
}
```

If we generalize the above to the case of matrices, all polynomial coefficients, as well as the x input and y output, are matrices.

The above code for evaluating a polynomial for a certain input works just as well for matrices, as long as the multiplication and addition operator are defined. So let's say we have a class `mat` and we have

```
mat::mat operator+( const mat& other ) const;
void mat::operator+=( const mat& other );
mat::mat operator*( const mat& other ) const;
void mat::operator*=( const mat& other );
```

Now we redefine the `polynomial` class, templated over the scalar type:

```
template< typename Scalar >
class polynomial {
private:
  vector<Scalar> coefficients;
public:
  polynomial( vector<Scalar> c )
    : coefficients(c) {};
  int degree() const { return coefficients.size()-1; };
  // 5 x^2 + 4 x + 3 = 5 x + 4 x + 3
  Scalar eval( Scalar x ) const {
    Scalar y{0.};
    for_each(coefficients.rbegin(),coefficients.rend(),
             [x,&y] (Scalar c) { y *= x; y += c; } );
    return y;
  };
```

The code using polynomials stays the same, except that we have to supply the scalar type as template parameter whenever we create a polynomial object. The above example of $p(x) = x^2 + 2$ becomes for scalars:

```
polynomial<double> x2p2( {2., 0., 1.} );
for ( auto x : {1., 2., 3.} ) {
  auto y = x2p2(x);
  cout << "Second power of x=" << x << " plus 2 gives y=" << y << '\n';
}
```

and for matrices:

```
polynomial<mat> x2p2( {2., 0., 1.} );
for ( auto x : {1., 2., 3.} ) {
  auto y = x2p2(x);
  cout << "Second power of x=" << x << " plus 2 gives y=" << y << '\n';
}
```

You see that after the templated definition the polynomial object is used entirely identically.

12.4 Concepts

The C++20 standard added the notion of *concept*.

Templates can be too generic. For instance, one could write a templated `gcd` function

```
template <typename T>
T gcd( T a, T b ) {
  if (b==0) return a;
  else return gcd(b,a%b);
}
```

which will work for various integral types. To prevent it being applied to non-integral types, you can specify a *concept* to the type:

```
template<typename T>
concept bool Integral() {
  return std::is_integral<T>::value;
}
```

used as:

```
template <typename T>
requires Integral<T>{}
T gcd( T a, T b ) { /* ... */ }
```

or

```
template <Integral T>
T gcd( T a, T b ) { /* ... */ }
```

Abbreviated function templates:

```
Integral auto gcd
    ( Integral auto a, Integral auto b )
  { /* ... */ }
```

Chapter 13

Error handling

13.1 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*: behavior 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 behavior:

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

See further section **??**.

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

13.2 Mechanisms to support error handling and debugging

13.2.1 Assertions

One way catch errors before they happen, is to sprinkle your code with assertions: statements about things that have to be true. For instance, if a function should mathematically always return a positive result, it may be a good idea to check for that anyway. You do this by using the `assert` command, which takes a boolean, and will stop your program if the boolean is false:

Check on valid input parameters:

```
#include <cassert>

// this function requires x<y
// it computes something positive
float f(x,y) {
  assert( x<y );
  return /* some result */;
}
```

Check on valid results:

```
float positive_outcome = f(x,y);
assert( positive_outcome>0 );
```

There is also `static_assert`, which checks compile-time conditions only.

Since checking an assertion is a minor computation, you may want to disable it when you run your program in production by defining the `NDEBUG` macro:

```
#define NDEBUG
```

One way of doing that is passing it as a compiler flag:

```
icpc -DNDEBUG yourprog.cpp
```

As an example of using assertions, we can consider the iteration function of the Collatz exercise **??**.

```
int collatz_next( int current ) {
  assert( current>0 );
  int next{-1};
  if (current%2==0) {
    next = current/2;
    assert(next<current);
  } else {
    next = 3*current+1;
    assert(next>current);
  }
  return next;
}
```

Remark 10 *If an assertion fails, your program will call* `std::abort`*. This is a less elegant exit than* `std::exit`*.*

13.2.2 Exception handling

Assertions are a little crude: they terminate your program, and the only thing you can do is debug, rewrite, and rerun. Some errors are of a type that you could possibly recover from them. In that case, exception are a better idea, since these can be handled inside the program.

Code with problem:

```
if ( /* some problem */ )
  throw(5);
  /* or: throw("error"); */
```

```
try {
  /* code that can go wrong */
} catch (...) { // literally three
    dots!
  /* code to deal with the problem
    */
}
```

13.2.2.1 Exception catching

During the run of your program, an error condition may occur, such as accessing a vector elements outside the bounds of that vector, that will make your program stop. You may see text on your screen

```
terminating with uncaught exception
```

The operative word here is *exception*: an exceptional situation that caused the normal program flow to have been interrupted. We say that your program *throws* an *exception*.

Code:

```
1 vector<float> x(5);
2 x.at(5) = 3.14;
```

Output
[except] boundthrow:

```
libc++abi.dylib: terminating
    with uncaught exception
    of type
    std::out_of_range: vector
```

Now you know that there is an error in your program, but you don't know where it occurs. You can find out, but trying to *catch the exception*.

Code:

```
1  vector<float> x(5);
2  for (int i=0; i<10; ++i) {
3    try {
4      x.at(i) = 3.14;
5    } catch (...) {
6      cout << "Exception indexing at: "
7           << i << '\n';
8      break;
9    }
10 }
```

Output
[except] catchbounds:

```
Exception indexing at: 5
```

13.2.2.2 Popular exceptions

- `std::out_of_range`: usually caused by using `at` with an invalid index.

13.2.2.3 Throw your own exceptions

Throwing an *exception* is one way of signaling an error or unexpected behavior:

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

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

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

If you're doing the prime numbers project, you can now do exercise 17.11.

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

```
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!

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 exceptions without specifying the type:

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

Exercise 13.1. 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.

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

```
f::f( int i )
  try : fbase(i) {
    // constructor body
  }
  catch (...) { // handle exception
  }
```

- 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 **??**.
- 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 { ... };
  ```

- There is no exception thrown when dereferencing a `nullptr`.

13.2.3 'Where does this error come from'

The CPP defines two macros, *__FILE__* and *__LINE__* that give you respectively the current file name and the current line number. You can use these to generate pretty error messages such as

```
Overflow occurred in line 25 of file numerics.cpp
```

The C++20 standard will offer `std::source_location` as a mechanism instead.

13.2.4 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.

13.2.5 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.

13.3 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 (see section 11.2), as well as the use of uninitialized data.

Chapter 14

Parallelism

In this chapter we briefly discuss shared memory parallelism, using two different mechanisms:

1. *OpenMP*, which is not part of the C++ standard, but is available under almost every compiler (the native *Apple Clang* being the exception), and which is widely used in scientific computing; and
2. *execution policies* in the C++17 standard.

For the theory of parallel computing, see HPC book [11], chapter 2; for more details on OpenMP, see MPI/OpenMP book [10], part II. Finally we note that we do not discuss other modes of parallelism, such as distributed memory computing (for which see MPI/OpenMP book [10], part I) or parallel computing with accelerators such as GPUs.

Remark 11 *Both of the parallelism systems discussed in this chapter are implemented through some form of threading. And threads are discussed in this course in chapter* **??**. *However, threads by themselves are typically not used in scientific computing. If that is your focus, OpenMP is at the same time easier to use, more powerful, and gives more control over issues relevant to parallel performance.*

14.1 Parallel loops

Probably the dominant form of parallelism in scientific computing is that of loops over large numbers of vector elements. Some examples are: the output components of a matrix-vector product, or any operation on the points of a Finite Elements (FE) grid.

OpenMP offers an easy way to execute such a loop in parallel: the programmer inserts a statement before the loop, stating that it can be executed in parallel. Note that it's up to the programmer to ensure that the loop is in fact parallel. Detecting parallelism is generally beyond the capabilities of a compiler.

14.1.1 Loops

The natural way to parallelize a loop in OpenMP is to use the `omp` pragma with the `parallel` and **`for`** directives. This causes OpenMP to spread the loop over the available threads and execute it in parallel:

```
#pragma omp parallel
#pragma omp for
for (int i=0; i<N; i++) {
  // do something with i
}
```

This fragment combines two OpenMP idioms:

1. First the `#parallel` directive creates a *team* of threads; after which
2. The `#for` directive is a *worksharing construct*: it divides the available work over the available threads.

You will often see the `parallel/for` directives combined:

```
#pragma omp parallel for
for ( /* stuff */ )
```

How does OpenMP know how parallel to run? It does not, for instance, detect the core count of your processor. Instead, you need to specify the number of threads explicitly, for instance through the `OMP_NUM_THREADS` environment variable. There are more environment variables, and options on the `parallel/for` directives, that you can use to fine-tune performance.

14.1.2 Reductions

After fully parallel loops, as just discussed, the next most common type of parallel operation is a reduction

```
for (int i=0; i<N; i++) {
  sum += x[i];
}
```

Parallelizing this as above would give a *race condition*: conflicts arising from the simultaneous access to `sum` by multiple threads. (For details, see HPC book [11], section 2.6.1.5.)

This is easily fixed (again: by you; the compiler does not alert you to this problem) by adding a `reduction` clause:

```
#pragma omp parallel
#pragma omp for reduction(+:sum)
for (int i=0; i<N; i++) {
  sum += x[i];
}
```

14.1.3 Parallel range-based loops

The above examples apply equally to OpenMP uses from C and from C++; in C++ we can be more elegant in the treatment of parallel loops by using range-based syntax.

Parallel loops in C++ can use range-based syntax as of OpenMP 5.0:

```
// vecdata.cxx
vector<float> values(100);

#pragma omp parallel for
for ( auto& elt : values ) {
  elt = 5.f;
}

float sum{0.f};
#pragma omp parallel for reduction(+:sum)
for ( auto elt : values ) {
```

```
    sum += elt;
  }
```

Tests not reported here show exactly the same speedup as the C code.

The C++20 *ranges* library is also supported:

```
// range.cxx
#     pragma omp parallel for reduction(+:count)
      for ( auto e : data )
        count += e;
#     pragma omp parallel for reduction(+:count)
      for ( auto e : data
              | std::ranges::views::drop(1) )
        count += e;
#     pragma omp parallel for reduction(+:count)
      for ( auto e : data
              | std::ranges::views::transform
              ( []( auto e ) { return 2*e; } ) )
        count += e;
```

```
==== Run range on 1 threads ====
sum of vector: 50000005000000 in 6.148
sum w/ drop 1: 50000004999999 in 6.017
sum times 2  : 100000010000000 in 6.012
==== Run range on 25 threads ====
sum of vector: 50000005000000 in 0.494
sum w/ drop 1: 50000004999999 in 0.477
sum times 2  : 100000010000000 in 0.489
==== Run range on 51 threads ====
sum of vector: 50000005000000 in 0.257
sum w/ drop 1: 50000004999999 in 0.248
sum times 2  : 100000010000000 in 0.245
==== Run range on 76 threads ====
sum of vector: 50000005000000 in 0.182
sum w/ drop 1: 50000004999999 in 0.184
sum times 2  : 100000010000000 in 0.185
==== Run range on 102 threads ====
sum of vector: 50000005000000 in 0.143
sum w/ drop 1: 50000004999999 in 0.139
sum times 2  : 100000010000000 in 0.134
==== Run range on 128 threads ====
sum of vector: 50000005000000 in 0.122
sum w/ drop 1: 50000004999999 in 0.11
sum times 2  : 100000010000000 in 0.106
 scaling results in: range-scaling-ls6.out
```

14.2 Reductions

Reduction can be applied to any class for which the reduction operator is defined as `operator+` or whichever operator the case may be.

```
// reductclass.cxx
class Thing {
private:
  float x;
public:
  Thing() : Thing( 0.f ) {};
  Thing( float x ) : x(x) {};
  Thing operator+( const Thing&
    other ) {
    return Thing( x + other.x );
  };
};
```

```
vector< Thing >
    things(500,Thing(1.f) );
Thing result(0.f);
#pragma omp parallel for reduction(
    +:result )
for ( const auto& t : things )
  result = result + t;
```

A default constructor is required for the internally used init value; see figure **??**.

Support for *C++ iterators*

```
#pragma omp declare reduction \
    (merge              // identifier
    : std::vector<int> // typelist
    : omp_out.insert(omp_out.end(), omp_in.begin(), omp_in.end()) // combiner
    )
```

In section **??** you saw how certain loop constructs can be realized in C++ through the *execution policy* argument of the standard algorithms. The same holds for reductions.

Use the `data` method to extract the array on which to reduce. Also, the reduction clause wants a variable, not an expression, for the array, so you need an extra bare pointer:

```
// reductarray.cxx
vector<int> data(nthreads,0);
int *datadata = data.data();
#pragma omp parallel for schedule(static,1) \
  reduction(+:datadata[:nthreads])
for (int it=0; it<nthreads; it++) {
  for (int i=0; i<nthreads; i++)
        datadata[i]++;
}
```

You can reduce with a templated function if you put both the declaration and the reduction in the same templated function:

```
template<typename T>
T generic_reduction( vector<T> tdata ) {
#pragma omp declare reduction                                    \
  (rwzt:T:omp_out=reduce_without_zero<T>(omp_out,omp_in))        \
  initializer(omp_priv=-1.f)

  T tmin = -1;
#pragma omp parallel for reduction(rwzt:tmin)
  for (int id=0; id<tdata.size(); id++)
    tmin = reduce_without_zero<T>(tmin,tdata[id]);
  return tmin;
```

```
};
```

which is then called with specific data:

```
auto tmin = generic_reduction<float>(fdata);
```

You can do a reduction over a `std::map` by merging thread-local maps:

```
// mapreduce.cxx
template<typename key>
class bincounter : public
    map<key,int> {
public:
// merge this with other map
  void operator+=( const
    bincounter<key>& other ) {
    for ( auto [k,v] : other )
      if (
    map<key,int>::contains(k) ) //
    c++20
        this->at(k) += v;
      else
        this->insert( {k,v} );
  };
// insert one char in this map
  void inc(char k) {
    if ( map<key,int>::contains(k) )
      this->at(k) += 1;
    else
      this->insert( {k,1} );
  };
};
```

```
using CharCounter =
    bincounter<char>;
#pragma omp declare reduction\
(       \
        +:CharCounter:omp_out +=
    omp_in \
        ) \
initializer( omp_priv =
    CharCounter{} )
CharCounter charcount;
#pragma omp parallel for
    reduction(+ : charcount)
for ( int i=0; i<text.size(); i++ )
  charcount.inc( text[i] );
```

You can use lambda expressions in the explicit expression:

```
// reductexpr.cxx
#pragma omp declare reduction\
  (minabs : int : \
   omp_out = \
       [] (int x,int y) -> int { \
          return abs(x) > abs(y) ? abs(y) : abs(x); } \
       (omp_in,omp_out) ) \
  initializer (omp_priv=limit::max())
```

You can not assign the lambda expression to a variable and use that, because `omp_in/out` are the only variables allowed in the explicit expression.

We will do this example only in C++ because of its ease of handling `std::vector`s.

```
// lockobject.cxx
class atomic_int {
private:
```

```
    omp_lock_t the_lock;
    int _value{0};
  public:
    atomic_int() {
      omp_init_lock(&the_lock);
    };
    atomic_int( const atomic_int& )
        = delete;
    atomic_int& operator=( const atomic_int& )
        = delete;
    ~atomic_int() {
      omp_destroy_lock(&the_lock);
    };
```

Running this:

```
atomic_int my_object;
vector<std::thread> threads;
for (int ithread=0; ithread<NTHREADS; ithread++) {
  threads.push_back
    ( std::thread(
      [=,&my_object] () {
        for (int iop=0; iop<nops; iop++)
          my_object += 1; } ) );
}
for ( auto &t : threads )
  t.join();
```

The use of `cout` may give jumbled output:lines can break at each <<.Use `stringstream` to form a single stream to output.

missing snippet [examples/omp/cxx/hello.cxx]hello-omp-cxx

OpenMP parallel regions can be in functions, including lambda expressions.

```
    const int s = [] () {
      int s;
#   pragma omp parallel
#   pragma omp master
      s = 2 * omp_get_num_threads();
      return s; }();
```

('Immediately Invoked Function Expression')

14.3 Execution policies

The C++17/C++20 standards have introduced the notion of *execution policy* to the 'standard algorithms', meaning the operations on containers that are in the `algorithm` library.

This parallelization is often done through Threading Building Blocks (Intel) (TBB).

As an example, let's consider prime number marking: create an array where `p[i]` is one if `i` is prime, zero otherwise.

```
#pragma omp parallel for schedule(guided,8)
 for (int i=0; i<nsize; i++) {
   results[i] = one_if_prime( number(i) );
 }

// primepolicy.cxx
transform( std::execution::par,
           numbers.begin(),numbers.end(),
           results.begin(),
           [] (int n ) -> int {
             return one_if_prime(n); }
           );
```

As a result we find that the parallel algorithm is competitive with OpenMP loop parallelization for low thread counts, but not higher.

```
==== Run primepolicy on 1 threads ====
OMP: found 0 primes; Time:      390 msec (threads= 1)
TBB: found 0 primes; Time:      392 msec
==== Run primepolicy on 25 threads ====
OMP: found 0 primes; Time:       17 msec (threads=25)
TBB: found 0 primes; Time:       19 msec
==== Run primepolicy on 51 threads ====
OMP: found 0 primes; Time:        9 msec (threads=51)
TBB: found 0 primes; Time:       13 msec
==== Run primepolicy on 76 threads ====
OMP: found 0 primes; Time:        6 msec (threads=76)
TBB: found 0 primes; Time:       15 msec
==== Run primepolicy on 102 threads ====
OMP: found 0 primes; Time:        5 msec (threads=102)
TBB: found 0 primes; Time:       71 msec
==== Run primepolicy on 128 threads ====
OMP: found 0 primes; Time:        4 msec (threads=128)
TBB: found 0 primes; Time:       55 msec
```

For reductions:

missing snippet reduceprimeomp missing snippet reduceprimecpp

```
==== Run reducepolicy on 1 threads ====
OMP: found 9592 primes; Time:      390 msec (threads= 1)
TBB: found 9592 primes; Time:      392 msec
==== Run reducepolicy on 25 threads ====
OMP: found 9592 primes; Time:       17 msec (threads=25)
TBB: found 9592 primes; Time:       20 msec
==== Run reducepolicy on 51 threads ====
OMP: found 9592 primes; Time:        8 msec (threads=51)
TBB: found 9592 primes; Time:       13 msec
==== Run reducepolicy on 76 threads ====
OMP: found 9592 primes; Time:        6 msec (threads=76)
TBB: found 9592 primes; Time:       23 msec
==== Run reducepolicy on 102 threads ====
OMP: found 9592 primes; Time:        5 msec (threads=102)
TBB: found 9592 primes; Time:      105 msec
==== Run reducepolicy on 128 threads ====
OMP: found 9592 primes; Time:        4 msec (threads=128)
TBB: found 9592 primes; Time:       54 msec
```

14.4 Thread-local data

A common problem in parallel codes is how to handle random number generation. In the case of OpenMP, the specific problem is that threads act as if they are created anew for each parallel region. Thus, to have a persistent random number generation we need thread-local data.

The new C++ *random* header has a threadsafe generator, by virtue of the statement in the standard that no STL object can rely on global state. The usual idiom can not be made threadsafe because of the initialization:

```
static random_device rd;
static mt19937 rng(rd);
```

However, the following works:

```
// privaterandom.cxx
static random_device rd;
static mt19937 rng;
#pragma omp threadprivate(rd)
#pragma omp threadprivate(rng)

int main() {

#pragma omp parallel
  rng = mt19937(rd());
```

You can then use the generator safely and independently:

```
#pragma omp parallel
  {
    stringstream res;
    uniform_int_distribution<int> percent(1, 100);
    res << "Thread " << omp_get_thread_num() << ": " << percent(rng) << "\n";
    cout << res.str();
  }
```

Chapter 15

Obscure stuff

15.1 Auto

This is not actually obscure, but it intersects many other topics, so we put it here for now.

15.1.1 Declarations

Sometimes the type of a variable is obvious:

```
std::vector< std::shared_ptr< myclass >>*
  myvar = new std::vector< std::shared_ptr< myclass >>
                ( 20, new myclass(1.3) );
```

(Pointer to vector of 20 shared pointers to `myclass`, initialized with unique instances.) You can write this as:

```
auto myvar =
  new std::vector< std::shared_ptr< myclass >>
            ( 20, new myclass(1.3) );
```

Return type can be deduced in C++17:

```
auto equal(int i,int j) {
  return i==j;
};
```

Return type of methods can be deduced in C++17:

```
class A {
private: float data;
public:
  A(float i) : data(i) {};
  auto &access() {
    return data; };
  void print() {
    cout << "data: " << data << '\n'; };
};
```

`auto` discards references and such:

Code:

```
A my_a(5.7);
auto get_data = my_a.access();
get_data += 1;
my_a.print();
```

Output
[auto] plainget:

```
data: 5.7
```

Combine `auto` and references:

Code:

```
A my_a(5.7);
auto &get_data = my_a.access();
get_data += 1;
my_a.print();
```

Output
[auto] refget:

```
data: 6.7
```

For good measure:

```
A my_a(5.7);
const auto &get_data = my_a.access();
get_data += 1; // WRONG does not compile
my_a.print();
```

15.1.2 Auto and function definitions

The return type of a function can be given with a *trailing return type* definition:

```
auto f(int i) -> double { /* stuff */ };
```

This notations is more common for lambdas, chapter 8.

15.1.3 `decltype`: declared type

There are places where you want the compiler to deduce the type of a variable, but where this is not immediately possible. Suppose that in

```
auto v = some_object.get_stuff();
f(v);
```

you want to put a `try ... catch` block around just the creation of v. This does not work:

```
try { auto v = some_object.get_stuff();
} catch (...) {}
f(v);
```

because the `try` block is a scope. It also doesn't work to write

```
auto v;
try { v = some_object.get_stuff();
} catch (...) {}
f(v);
```

because there is no indication what type `v` is created with.

Instead, it is possible to query the type of the expression that creates `v` with `decltype`:

```
decltype(some_object.get_stuff()) v;
try { auto v = some_objects.get_stuff();
} catch (...) {}
f(v);
```

??

15.2 Casts

In C++, constants and variables have clear types. For cases where you want to force the type to be something else, there is the *cast* mechanism. With a cast you tell the compiler: treat this thing as such-and-such a type, no matter how it was defined.

In C, there was only one casting mechanism:

```
sometype x;
othertype y = (othertype)x;
```

This mechanism is still available as the **`reinterpret_cast`**, which does 'take this byte and pretend it is the following type':

```
sometype x;
auto y = reinterpret_cast<othertype>(x);
```

The inheritance mechanism necessitates another casting mechanism. An object from a derived class contains in it all the information of the base class. It is easy enough to take a pointer to the derived class, the bigger object, and cast it to a pointer to the base object. The other way is harder.

Consider:

```
class Base {};
class Derived : public Base {};
Base *dobject = new Derived;
```

Can we now cast dobject to a pointer-to-derived ?

- **`static_cast`** assumes that you know what you are doing, and it moves the pointer regardless.
- **`dynamic_cast`** checks whether `dobject` was actually of class `Derived` before it moves the pointer, and returns **`nullptr`** otherwise.

Remark 12 *One further problem with the C-style casts is that their syntax is hard to spot, for instance by searching in an editor. Because C++ casts have a unique keyword, they are easier to recognize in a text editor.*

Further reading https://www.quora.com/How-do-you-explain-the-differences-among-static_cast-reinterpret_cast-const_cast-and-dynamic_cast-to-a-new-C++-programmer/answer/Brian-Bi

15.2.1 Static cast

One use of casting is to convert to constants to a 'larger' type. For instance, allocation does not use integers but *size_t*.

```
int hundredk = 100000;
int overflow;
overflow = hundredk*hundredk;
cout << "overflow: " << overflow << '\n';
size_t bignumber = static_cast<size_t>(hundredk)*hundredk;
cout << "bignumber: " << bignumber << '\n';
```

However, if the conversion is possible, the result may still not be 'correct'.

Code:

```
1 long int hundredg = 100000000000;
2 cout << "long number:      "
3      << hundredg << '\n';
4 int overflow;
5 overflow = static_cast<int>(hundredg);
6 cout << "assigned to int: "
7      << overflow << '\n';
```

Output

```
[cast] intlong:
```

There are no runtime tests on static casting.

Static casts are a good way of casting back void pointers to what they were originally.

15.2.2 Dynamic cast

Consider the case where we have a base class and derived classes.

```
class Base {
public:
  virtual void print() = 0;
};
class Derived : public Base {
public:
  virtual void print() {
    cout << "Construct derived!"
         << '\n'; };
};
class Erived : public Base {
public:
  virtual void print() {
    cout << "Construct erived!"
         << '\n'; };
};
```

Also suppose that we have a function that takes a pointer to the base class:

```
void f( Base *obj ) {
  Derived *der =
    dynamic_cast<Derived*>(obj);
  if (der==nullptr)
    cout << "Could not be cast to Derived"
```

```
        << '\n';
  else
    der->print();
};
```

The function can discover what derived class the base pointer refers to:

```
Base *object = new Derived();
f(object);
Base *nobject = new Erived();
f(nobject);
```

If we have a pointer to a derived object, stored in a pointer to a base class object, it's possible to turn it safely into a derived pointer again:

Code:

```
1 Base *object = new Derived();
2 f(object);
3 Base *nobject = new Erived();
4 f(nobject);
```

Output
[cast] deriveright:

```
Construct derived!
Could not be cast to Derived
```

On the other hand, a `static_cast` would not do the job:

Code:

```
1 void g( Base *obj ) {
2   Derived *der =
3     static_cast<Derived*>(obj);
4   der->print();
5 };
6     /* ... */
7     Base *object = new Derived();
8     g(object);
9     Base *nobject = new Erived();
10    g(nobject);
```

Output
[cast] derivewrong:

```
Construct derived!
Construct erived!
```

Note: the base class needs to be polymorphic, meaning that that pure virtual method is needed. This is not the case with a static cast, but, as said, this does not work correctly in this case.

15.2.3 Const cast

With `const_cast` you can add or remove const from a variable. This is the only cast that can do this.

15.2.4 Reinterpret cast

The `reinterpret_cast` is the crudest cast, and it corresponds to the C mechanism of 'take this byte and pretend it of type whatever'. There is a legitimate use for this:

```
void *ptr;
ptr = malloc( how_much );
auto address = reinterpret_cast<long int>(ptr);
```

so that you can do arithmetic on the address. For this particular case, `intptr_t` is actually better.

15.2.5 A word about void pointers

A traditional use for casts in C was the treatment of *void pointer*s. The need for this is not as severe in C++ as it was before.

A typical use of void pointers appears in the PETSc [4, 5] library. Normally when you call a library routine, you have no further access to what happens inside that routine. However, PETSc has the functionality for you to specify a monitor so that you can print out internal quantities.

```
int KSPSetMonitor(KSP ksp,
  int (*monitor)(KSP,int,PetscReal,void*),
  void *context,
  // one parameter omitted
  );
```

Here you can declare your own monitor routine that will be called internally: the library makes a *callback* to your code. Since the library can not predict whether your monitor routine may need further information in order to function, there is the `context` argument, where you can pass a structure as void pointer.

This mechanism is no longer needed in C++ where you would use a *lambda* (chapter 8):

```
KSPSetMonitor( ksp,
  [mycontext] (KSP k,int ,PetscReal r) -> int {
    my_monitor_function(k,r,mycontext); } );
```

15.3 Fine points of scalar types

15.3.1 Integers

There are several integer types, differing by the number of bytes they take up, and by whether they are signed or not. Here is a systematic discussion.

15.3.1.1 The long and short of it

In addition to `int`, there are also `short`, `long` and `long long` integers. These names give some, imperfect, indication as to their size, and therefore range. The next section will discuss types with a more precise definition.

- A short int is at least 16 bits;
- An integer is at least 16 bits, which was the case in the old days of the *DEC PDP-11*, but nowadays they are commonly 32 bits;
- A long integer is at least 32 bits, but often 64; the big exception being *Windows* where `long` integers are 32 bits.
- A 'long long' integer is at least 64 bits.
- If you need only one byte for your integer, you can use a `char`; see section **??**.

There are a number of generally accepted *data models* for the definition of these types; see HPC book [11], section 3.7.1.

All these types are signed integers: they accomodate a range $-N \ldots 0 \ldots N+1$ for some N. Prefixing these types with the keyword `unsigned` gives nonnegative types with a range $0 \ldots 2N$.

If you want to determine precisely what the range of integers or real numbers is that is stored in an `int` variable and such, you can use the `limits` header; see section 3.4.

15.3.1.2 Byte by byte

If you want to dictate how many bits to use, there is the `cstdint` header, which defines such types as `int16_t` and `uint16_t`:

```
int8_t   // 8 bits
uint8_t  // 8 bits, unsigned
int16_t  // 16 bits
uint16_t // 16 bits, unsigned
int64_t  // 64 bits
uint64_t // 64 bits, unsigned
```

Code:

```
cout << "max int      : "
     << numeric_limits<int>::max() <<
    '\n';
cout << "max unsigned: "
     << numeric_limits<unsigned
    int>::max() << '\n';
```

Output
[int] limit:

```
max int     : 2147483647
max unsigned: 4294967295
```

(For the mechanism used here, see section 3.4.)

15.3.1.3 Integer overflow

From the limited space that an integer takes, it is clear that there have to be a smallest and largest integer. Querying such limits on integers is discussed in section 3.4.

Computations that exceed those limits have an undefined result; however, since C++20 integers are guaranteed to be stored as 'two's complement'; see HPC book [11], section 3.2.

15.3.1.4 The dangers of unsigned integers

Unsigned values are fraught with danger. For instance, comparing them to integers gives counter-intuitive results:

Code:

```
unsigned int one{1};
int mone{-1};
cout << "less: " << boolalpha <<
    (mone<one) << '\n';
```

Output
[int] cmp:

```
less: false
```

For this reason, C++20 has introduced utility functions `cmp_equal` and such (in the `utility` header) that do these comparisons correctly.

15.3.1.5 Why different integers?

Historically, short integers were motivated by a need to save space. These days, that argument is largely irrelevant. The argument for short integers now derives from the limited *bandwidth* of processors: 'streaming' type applications are determined by the available bandwidth, and using short integers effectively doubles that bandwidth.

Long integers are also motivated by the current abundance of memory and disc space. Modern applications have large amounts of memory available to them, more than can be addressed with a 32-bit integer. This phenomenon is exacerbated by the existence of *multicore* processors, that can handle many times the memory of earlier processors in the same time.

15.3.2 Floating point types

Truncation and precision are tricky things. As a small illustration, let's do the same computation in single and double precision. While the results show the same with the default `cout` formatting, if we subtract them we see a non-zero difference.

Code:

```
double point3d = .3/.7;
float
  point3f = .3f/.7f,
  point3t = point3d;
cout << "double precision: "
     << point3d << '\n'
     << "single precision: "
     << point3f << '\n'
     << "difference with truncation:"
     << point3t - point3f
     << '\n';
```

Output
[basic] point3:

```
double precision: 0.428571
single precision: 0.428571
difference with
    truncation:-2.98023e-08
```

You can actually explain the size of this difference, however, we defer discussion of the details of floating point arithmetic to HPC book [11], chapter 3.

15.3.3 Not-a-number

The *IEEE 754* standard for floating point numbers states that certain bit patterns correspond to the value `NaN`: 'not a number'. This is the result of such computations as the square root of a negative number, or zero divided by zero; you can also explicitly generate it with `quiet_NaN` or `signalling_NaN`.

- `NaN` is only defined for floating point types: the test `has_quiet_NaN` is false for other types such as `bool` or `int`.
- Even through `complex` is built on top of floating point types, there is no `NaN` for it.

Code:

```
cout << "Double NaNs: "
     <<
     std::numeric_limits<double>::quiet_NaN(
        << ' ' // nan
     <<
     std::numeric_limits<double>::signaling_NaN()
     << ' ' // nan
     << '\n'
     << "zero divided by zero: "
     << 0 / 0.0          << '\n';
cout << boolalpha
     << "Int has NaN: "
     <<
     std::numeric_limits<int>::has_quiet_NaN
     << '\n';
```

Output
[limits] nan:

```
Double NaNs: nan nan
zero divided by zero: nan
Int has NaN: false
```

15.3.3.1 Tests

There are tests for detecting whether a number is `Inf` or `NaN`. However, using these may slow a computation down.

The functions `isinf` and `isnan` are defined for the floating point types (`float`, `double`, `long double`), returning a `bool`.

```
#include <math.h>
isnan(-1.0/0.0);   // false
isnan(sqrt(-1.0)); // true
isinf(-1.0/0.0);   // true
isinf(sqrt(-1.0)); // false
```

15.3.4 Common numbers

```
#include <numbers>
static constexpr float pi = std::numbers::pi;
```

15.4 lvalue vs rvalue

The terms 'lvalue' and 'rvalue' sometimes appear in compiler error messages.

```
int foo() {return 2;}

int main()
{
    foo() = 2;

    return 0;
}
```

```
# gives:
test.c: In function 'main':
test.c:8:5: error: lvalue required as left operand of assignment
```

See the 'lvalue' and 'left operand'? To first order of approximation you're forgiven for thinking that an *lvalue* is something on the left side of an assignment. The name actually means 'locator value': something that's associated with a specific location in memory. Thus an lvalue is, also loosely, something that can be modified.

An *rvalue* is then something that appears on the right side of an assignment, but is really defined as everything that's not an lvalue. Typically, rvalues can not be modified.

The assignment `x=1` is legal because a variable `x` is at some specific location in memory, so it can be assigned to. On the other hand, `x+1=1` is not legal, since `x+1` is at best a temporary, therefore not at a specific memory location, and thus not an lvalue.

Less trivial examples:

```
int foo() { x = 1; return x; }
int main() {
  foo() = 2;
}
```

is not legal because `foo` does not return an lvalue. However,

```
class foo {
private:
  int x;
public:
  int &xfoo() { return x; };
};
int main() {
  foo x;
  x.xfoo() = 2;
```

is legal because the function `xfoo` returns a reference to the non-temporary variable `x` of the `foo` object.

Not every lvalue can be assigned to: in

```
const int a = 2;
```

the variable `a` is an lvalue, but can not appear on the left hand side of an assignment.

15.4.1 Conversion

Most lvalues can quickly be converted to rvalues:

```
int a = 1;
int b = a+1;
```

Here `a` first functions as lvalue, but becomes an rvalue in the second line.

The ampersand operator takes an lvalue and gives an rvalue:

```
int i;
int *a = &i;
&i = 5; // wrong
```

15.4.2 References

The ampersand operator yields a reference. It needs to be assigned from an lvalue, so

```
std::string &s = std::string(); // wrong
```

is illegal. The type of `s` is an 'lvalue reference' and it can not be assigned from an rvalue.

On the other hand

```
const std::string &s = std::string();
```

works, since `s` can not be modified any further.

15.4.3 Rvalue references

A new feature of C++ is intended to minimize the amount of data copying through *move semantics*.

Consider a copy assignment operator

```
BigThing& operator=( const BigThing &other ) {
  BigThing tmp(other); // standard copy
  std::swap( /* tmp data into my data */ );
  return *this;
};
```

This calls a copy constructor and a destructor on `tmp`. (The use of a temporary makes this safe under exceptions. The `swap` method never throws an exception, so there is no danger of half-copied memory.)

However, if you assign

```
thing = BigThing(stuff);
```

Now a constructor and destructor is called for the temporary rvalue object on the right-hand side.

Using a syntax that is new in *C++*, we create an *rvalue reference*:

```
BigThing& operator=( BigThing &&other ) {
  swap( /* other into me */ );
  return *this;
}
```

15.5 Move semantics

With an *overloaded operator*, such as addition, on matrices (or any other big object):

```
Matrix operator+(Matrix &a,Matrix &b);
```

the actual addition will involve a copy:

```
Matrix c = a+b;
```

Use a move constructor:

```
class Matrix {
private:
  Representation rep;
public:
  Matrix(Matrix &&a) {
    rep = a.rep;
    a.rep = {};
  }
};
```

15.6 Graphics

C++ has no built-in graphics facilities, so you have to use external libraries such as *OpenFrameworks*, https://openframeworks.cc.

15.7 Standards timeline

Each standard has many changes over the previous.

If you want to detect what language standard you are compiling with, use the `__cpluscplus` macro:

Code:

```
cout << "C++ version: " << __cplusplus
     << '\n';
```

Output

```
[basic] version:
C++ version: 201703
```

This returns a `long int` with possible values `199711`, `201103`, `201402`, `201703`, `202002`.

Here are some of the highlights of the various standards.

15.7.1 C++98/C++03

Of the C++03 standard we only highlight deprecated features.

- `auto_ptr` was an early attempt at smart pointers. It is deprecated, and C++17 compilers will actually issue an error on it. For current smart pointers see chapter 11.

15.7.2 C++11

- `auto`

  ```
  const auto count = std::count
      (begin(vec),end(vec),value);
  ```

  The `count` variable now gets the type of whatever `vec` contained.
- Range-based for. We have been treating this as the base case, for instance in section **??**. The pre-C++11 mechanism, using an *iterator* (section 9.2.1) is largely obviated.
- Lambdas. See chapter 8.
- Chrono; see section 41.1.

- Variadic templates.
- Smart pointers; see chapter 11.

```
unique_ptr<int> iptr( new int(5) );
```

  This fixes problems with *auto_ptr*.
- *constexpr*

```
constexpr int get_value() {
  return 5*3;
}
```

15.7.3 C++14

C++14 can be considered a bug fix on C++11. It simplifies a number of things and makes them more elegant.

- Auto return type deduction:

```
auto f() {
  SomeType something;
  return something;
}
```

- Generic lambdas (section 8.3.2)

```
const auto count = std::count(begin(vec),end(vec),
    [] ( const auto i ) { return i<3; }
);
```

  Also more sophisticated capture expressions.
- *constexpr*

```
constexpr int get_value() {
  int val = 5;
  int val2 = 3;
  return val*val2
}
```

15.7.4 C++17

- Optional; section **??**.
- Structured binding declarations as an easier way of dissecting tuples; section **??**.
- Init statement in conditionals; section 2.6.1.

15.7.5 C++20

- *modules*: these offer a better interface specification than using *header files*.
- *coroutines*, another form of parallelism.
- *concepts* including in the standard library via ranges; section 12.4.
- *spaceship operator* including in the standard library
- broad use of normal C++ for direct compile-time programming, without resorting to template meta programming (see last trip reports)
- *ranges*

- *calendars* and *time zones*
- *text formatting*
- `span`. See section **??**.
- `numbers`. Section 15.3.4.
- Safe integer/unsigned comparison; section 15.3.1.2; integers are guaranteed two's complement.

Here is a summary with examples: https://oleksandrkvl.github.io/2021/04/02/cpp-20-overview.html.

15.7.6 C++23

- `mdspan` offers multi-dimensional array; this is an extension of the C++17 `span` mechanism.

PART II

EXERCISES AND PROJECTS

Chapter 16

Style guide for project submissions

The purpose of computing is insight, not numbers. (Richard Hamming)

Your project writeup is equally important as the code. Here are some common-sense guidelines for a good writeup. However, not all parts may apply to your project. Use your good judgment.

16.1 General approach

As a general rule, consider programming as an experimental science, and your writeup as a report on some tests you have done: explain the problem you're addressing, your strategy, your results.

Turn in a writeup in pdf form (Word and text documents are not acceptable) that was generated from a text processing program such (preferably) LaTeX. For a tutorial, see Tutorials book [9], section-15.

16.2 Style

Your report should obey the rules of proper English.

- Observing correct spelling and grammar goes without saying.
- Use full sentences.
- Try to avoid verbiage that is disparaging or otherwise inadvisable. The *Google developer documentation style guide* [13] is a great resource.

16.3 Structure of your writeup

Consider this project writeup an opportunity to practice writing a scientific article.

Start with the obvious stuff.

- Your writeup should have a title. Not 'Project', but something like 'Simulation of Chronosynclastic Enfundibula'.
- Author and contact information. This differs per case. Here it is: your name, EID, TACC username, and email.
- Introductory section that is extremely high level: what is the problem, what did you do, what did you find.

- Conclusion: what do your findings mean, what are limitations, opportunities for future extensions.
- Bibliography.

16.3.1 Introduction

The reader of your document need not be familiar with the project description, or even the problem it addresses. Indicate what the problem is, give theoretical background if appropriate, possibly sketch a historic background, and describe in global terms how you set out to solve the problem, as well as a brief statement of your findings.

16.3.2 Detailed presentation

See section 16.5 below.

16.3.3 Discussion and summary

Separate the detailed presentation from a discussion at the end: you need to have a short final section that summarizes your work and findings. You can also discuss possible extensions of your work to cases not covered.

16.4 Experiments

You should not expect your program to run once and give you a final answer to your research question.

Ask yourself: what parameters can be varied, and then vary them! This allows you to generate graphs or multi-dimensional plots.

If you vary a parameter, think about what granularity you use. Do ten data points suffice, or do you get insight from using $10,000$?

Above all: computers are very fast, they do a billion operations per second. So don't be shy in using long program runs. Your program is not a calculator where a press on the button immediately gives the answer: you should expect program runs to take seconds, maybe minutes.

16.5 Detailed presentation of your work

The detailed presentation of your work is as combination of code fragments, tables, graphs, and a description of these.

16.5.1 Presentation of numerical results

You can present results as graphs/diagrams or tables. The choice depends on factors such as how many data points you have, and whether there is an obvious relation to be seen in a graph.

Graphs can be generated any number of ways. Kudos if you can figure out the LaTeX `tikz` package, but Matlab or Excel are acceptable too. No screenshots though.

Number your graphs/tables and refer to the numbering in the text. Give the graph a clear label and label the axes.

16.5.2 Code

Your report should describe in a global manner the algorithms you developed, and you should include relevant code snippets. If you want to include full listings, relegate that to an appendix: code snippets in the text should only be used to illustrate especially salient points.

Do not use screen shots of your code: at the very least use a monospace font such as the `verbatim` environment, but using the `listings` package (used in this book) is very much recommended.

Chapter 17

Prime numbers

In this chapter you will do a number of exercises regarding prime numbers that build on each other. Each section lists the required prerequisites. Conversely, the exercises here are also referenced from the earlier chapters.

17.1 Arithmetic

Before doing this section, make sure you study section **??**.

Exercise 17.1. Read two numbers and print out their modulus. The modulus operator is `x%y`.

- Can you also compute the modulus without the operator?
- What do you get for negative inputs, in both cases?
- Assign all your results to a variable before outputting them.

17.2 Conditionals

Before doing this section, make sure you study section **??**.

Exercise 17.2. Read two numbers and print a message stating whether the second is as divisor of the first:

Code:

```
int number,divisor;
bool is_a_divisor;
  /* ... */
if (
  /* ... */
    ) {
  cout << "Indeed, " << divisor
       << " is a divisor of "
       << number << '\n';
} else {
  cout << "No, " << divisor
       << " is not a divisor of "
       << number << '\n';
}
```

Output

[primes] division:

```
( echo 6 ; echo 2 ) |
   divisiontest
Enter a number:
Enter a trial divisor:
Indeed, 2 is a divisor of 6

( echo 9 ; echo 2 ) |
   divisiontest
Enter a number:
Enter a trial divisor:
No, 2 is not a divisor of 9
```

17.3 Looping

Before doing this section, make sure you study section **??**.

Exercise 17.3. Read an integer and set a boolean variable to determine whether it is prime by testing for the smaller numbers if they divide that number.

Print a final message

```
Your number is prime
```

or

```
Your number is not prime: it is divisible by ....
```

where you report just one found factor.

Printing a message to the screen is hardly ever the point of a serious program. In the previous exercise, let's therefore assume that the fact of primeness (or non-primeness) of a number will be used in the rest of the program. So you want to store this conclusion.

Exercise 17.4. Rewrite the previous exercise with a boolean variable to represent the primeness of the input number.

Exercise 17.5. Read in an integer r. If it is prime, print a message saying so. If it is not prime, find integers $p \leq q$ so that $r = p \cdot q$ and so that p and q are as close together as possible. For instance, for $r = 30$ you should print out `5,6`, rather than `3,10`. You are allowed to use the function `sqrt`.

17.4 Functions

Before doing this section, make sure you study section **??**.

chapter]ch:function

Above you wrote several lines of code to test whether a number was prime. Now we'll turn this code into a function.

Exercise 17.6. Write a function `is_prime` that has an integer parameter, and returns a boolean corresponding to whether the parameter was prime.

```
int main() {
  bool isprime;
  isprime = is_prime(13);
```

Write a main program that reads the number in, and prints the value of the boolean. (How is the boolean rendered? See section 7.1.2Boolean outputsubsection.7.1.2.)

Does your function have one or two `return` statements? Can you imagine what the other possibility looks like? Do you have an argument for or against it?

17.5 While loops

Before doing this section, make sure you study section **??**.

Exercise 17.7. Take your prime number testing function `is_prime`, and use it to write a program that prints multiple primes:

- Read an integer `how_many` from the input, indicating how many (successive) prime numbers should be printed.
- Print that many successive primes, each on a separate line.
- (Hint: keep a variable `number_of_primes_found` that is increased whenever a new prime is found.)

17.6 Classes and objects

Before doing this section, make sure you study section **??**.

Exercise 17.8. Write a class `primegenerator` that contains:

- Methods `number_of_primes_found` and `nextprime`;
- Also write a function `isprime` that does not need to be in the class.

Your main program should look as follows:

```
cin >> nprimes;
primegenerator sequence;
while (sequence.number_of_primes_found()<nprimes) {
  int number = sequence.nextprime();
  cout << "Number " << number << " is prime" << '\n';
}
```

In the previous exercise you defined the `primegenerator` class, and you made one object of that class:

```
primegenerator sequence;
```

But you can make multiple generators, that all have their own internal data and are therefore independent of each other.

Exercise 17.9. The *Goldbach conjecture* says that every even number, from 4 on, is the sum of two primes $p + q$. Write a program to test this for the even numbers up to a bound that you read in. Use the `primegenerator` class you developed in exercise 17.8Classes and objectsexcounter.17.8.

This is a great exercise for a top-down approach!

1. Make an outer loop over the even numbers e.
2. For each e, generate all primes p.
3. From $p + q = e$, it follows that $q = e - p$ is prime: test if that q is prime.

For each even number `e` then print `e`, `p`, `q`, for instance:

```
The number 10 is 3+7
```

If multiple possibilities exist, only print the first one you find.

An interesting corollary of the Goldbach conjecture is that each prime (start at 5) is equidistant between two other primes.

The *Goldbach conjecture* says that every even number $2n$ (starting at 4), is the sum of two primes $p + q$:

$$2n = p + q.$$

Equivalently, every number n is equidistant from two primes:

$$n = \frac{p+q}{2} \qquad \text{or} \qquad q - n = n - p.$$

In particular this holds for each prime number:

$$\forall_{r \text{ prime}} \exists_{p,q \text{ prime}} : r = (p+q)/2 \text{ is prime.}$$

Exercise 17.10.

Write a program that tests this. You need at least one loop that tests all primes r; for each r you then need to find the primes p, q that are equidistant to it. Do you use two generators for this, or is one enough? Do you need three, for p, q, r?

For each r value, when the program finds the p, q values, print the p, q, r triple and move on to the next r.

17.6.1 Exceptions

Before doing this section, make sure you study section 13.2.2.

Exercise 17.11. Revisit the prime generator class (exercise 17.8Classes and objectsexcounter.17.8) and let it throw an exception once the candidate number is too large. (You can hardwire this maximum, or use a limit; section 3.4Limitssection.3.4.)

Code:

```
try {
  do {
    auto cur = primes.nextprime();
    cout << cur << '\n';
  } while (true);
} catch ( string s ) {
  cout << s << '\n';
}
```

Output
[primes] genx:

```
9931
9941
9949
9967
9973
Reached max int
```

17.6.2 Prime number decomposition

Before doing this section, make sure you study section **??**.

Design a class `Integer` which stores its value as its prime number decomposition. For instance,

$$180 = 2^2 \cdot 3^3 \cdot 5 \qquad \Rightarrow \qquad \texttt{[2:2, 3:2, 5:1]}$$

You can implement this decomposition itself as a *vector*, (the i-th location stores the exponent of the i-th prime) but let's use a `map` instead.

Exercise 17.12. Write a constructor of an `Integer` from an `int`, and methods `as_int` / `as_string` that convert the decomposition back to something classical. Start by assuming that each prime factor appears only once.

Code:

```
Integer i2(2);
cout << i2.as_string() << ": "
     << i2.as_int() << '\n';

Integer i6(6);
cout << i6.as_string() << ": "
     << i6.as_int() << '\n';
```

Output
[primes] decomposition26:

```
2^1 : 2
2^1 3^1 : 6
```

Exercise 17.13. Extend the previous exercise to having multiplicity > 1 for the prime factors.

Code:

```
Integer i180(180);
cout << i180.as_string() << ": "
     << i180.as_int() << '\n';
```

Output
[primes] decomposition180:

```
2^2 3^2 5^1 : 180
```

Implement addition and multiplication for `Integer`s.

Implement a class `Rational` for rational numbers, which are implemented as two `Integer` objects. This class should have methods for addition and multiplication. Write these through operator overloading if you've learned this.

Make sure you always divide out common factors in the numerator and denominator.

17.7 Ranges

Before doing this section, make sure you study section 9.1.

Exercise 17.14. Write a range-based code that tests

$$\forall_{\text{prime } p}: \exists_{\text{prime } q}: q > p$$

Exercise 17.15. Rewrite exercise 17.10, using only range expressions, and no loops.

Exercise 17.16. In the above Goldbach exercises you probably needed two prime number sequences, that, however, did not start at the same number. Can you make it so that your code reads

```
all_of( primes_from(5) /* et cetera */
```

17.8 Other

The following exercise requires `std::optional`, which you can learn about in section **??**.

Exercise 17.17. Write a function `first_factor` that optionally returns the smallest factor of a given input.

```
auto factor = first_factor(number);
if (factor.has_value())
  cout << "Found factor: " << factor.value() << '\n';
```

17.9 Eratosthenes sieve

The Eratosthenes sieve is an algorithm for prime numbers that step-by-step filters out the multiples of any prime it finds.

1. Start with the integers from 2: $2, 3, 4, 5, 6, \ldots$
2. The first number, 2, is a prime: record it and remove all multiples, giving

   $$3, 5, 7, 9.11, 13, 15, 17 \ldots$$

3. The first remaining number, 3, is a prime: record it and remove all multiples, giving

   $$5, 7, 11, 13, 17, 19, 23, 25, 29 \ldots$$

4. The first remaining number, 5, is a prime: record it and remove all multiples, giving

   $$7, 11, 13, 17, 19, 23, 29, \ldots$$

17.9.1 Arrays implementation

The sieve can be implemented with an array that stores all integers.

Exercise 17.18. Read in an integer that denotes the largest number you want to test. Make an array of integers that long. Set the elements to the successive integers. Apply the sieve algorithm to find the prime numbers.

17.9.2 Streams implementation

The disadvantage of using an array is that we need to allocate an array. What's more, the size is determined by how many integers we are going to test, not how many prime numbers we want to generate. We are going to take the idea above of having a generator object, and apply that to the sieve algorithm: we will now have multiple generator objects, each taking the previous as input and erasing certain multiples from it.

Exercise 17.19. Write a `stream` class that generates integers and use it through a pointer.

Code:

```
for (int i=0; i<7; ++i)
  cout << "Next int: "
       << the_ints->next() << '\n';
```

Output
[sieve] ints:

```
Next int: 2
Next int: 3
Next int: 4
Next int: 5
Next int: 6
Next int: 7
Next int: 8
```

Next, we need a stream that takes another stream as input, and filters out values from it.

Exercise 17.20. Write a class `filtered_stream` with a constructor

```
filtered_stream(int filter,shared_ptr<stream> input);
```

that

1. Implements `next`, giving filtered values,
2. by calling the `next` method of the input stream and filtering out values.

Code:

```
auto integers =
  make_shared<stream>();
auto odds =
  shared_ptr<stream>
  ( new filtered_stream(2,integers) );
for (int step=0; step<5; ++step)
  cout << "next odd: "
       << odds->next() << '\n';
```

Output
[sieve] odds:

```
next odd: 3
next odd: 5
next odd: 7
next odd: 9
next odd: 11
```

Now you can implement the Eratosthenes sieve by making a `filtered_stream` for each prime number.

Exercise 17.21. Write a program that generates prime numbers as follows.

- Maintain a `current` stream, that is initially the stream of prime numbers.
- Repeatedly:

- Record the first item from the current stream, which is a new prime number;
- and set `current` to a new stream that takes `current` as input, filtering out multiples of the prime number just found.

17.10 Range implementation

Before doing this section, make sure you study section 9.4.2.

If we write the prime number definition

$$\begin{array}{ll} D(n,d) & \equiv n|d = 0 \\ P(n) & \equiv \forall_{d\leq\sqrt{N}}: \neg D(n,d) \end{array}$$

we see that this involves two streams that we iterate over:

1. First there is the set of all d such that $d^2 \leq n$; then
2. We have the set of booleans testing whether these d values are divisors.

Exercise 17.22. Use the `iota` range view to generate all integers from 2 to infinity, and find a range view that cuts off the seqwuence at the last possible divisor.

Then use the `all_of` or `any_of` rangified algorithms to test whether any of these potential divisors are actually a divisor, and therefore whether or not your number is prime.

Exercise 17.23. Use the `filter` view to filter from an `iota` view those elements that are prime.

Exercise 17.24. Make a `primes` class that can be ranged:

Code:

```
primegenerator allprimes;
for ( auto p : allprimes ) {
  cout << p << ", ";
  if (p>100) break;
}
cout << '\n';
```

Output

```
[primes] range:

missing snippet
../code/primes/range.runout
```

17.11 User-friendliness

Use the `cxxopts` package (section 35.2Options processing: `cxxopts`section.35.2) to add commandline options to some primality programs.

Exercise 17.25. Take your old prime number testing program, and add commandline options:

- the `-h` option should print out usage information;
- specifying a single int `--test 1001` should print out all primes under that number;
- specifying a set of ints `--tests 57,125,1001` should test primeness for those.

Chapter 18

Geometry

In this set of exercises you will write a small 'geometry' package: code that manipulates points, lines, shapes. These exercises mostly use the material of section **??**.

18.1 Basic functions

Exercise 18.1. Write a function with (float or double) inputs x, y that returns the distance of point (x, y) to the origin.

Test the following pairs: $1, 0$; $0, 1$; $1, 1$; $3, 4$.

Exercise 18.2. Write a function with inputs x, y, θ that alters x and y corresponding to rotating the point (x, y) over an angle θ.

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

Your code should behave like:

Code:

```
const float pi = 2*acos(0.0);
float x{1.}, y{0.};
rotate(x,y,pi/4);
cout << "Rotated halfway: ("
     << x << "," << y << ")" << '\n';
rotate(x,y,pi/4);
cout << "Rotated to the y-axis: ("
     << x << "," << y << ")" << '\n';
```

Output

```
[geom] rotate:
Rotated halfway:
    (0.707107,0.707107)
Rotated to the y-axis: (0,1)
```

18.2 Point class

Before doing this section, make sure you study section **??**.

A class can contain elementary data. In this section you will make a `Point` class that models Cartesian coordinates and functions defined on coordinates.

Exercise 18.3. Make class `Point` with a constructor

```
Point( float xcoordinate, float ycoordinate );
```

Write the following methods:

- `distance_to_origin` returns a `float`.
- `angle` computes the angle of vector (x, y) with the x-axis.

Exercise 18.4. Extend the `Point` class of the previous exercise with a method: `distance` that computes the distance between this point and another: if `p,q` are `Point` objects,

```
p.distance(q)
```

computes the distance between them.

Exercise 18.5. Write a method *halfway* that, given two `Point` objects `p,q`, construct the `Point` halfway, that is, $(p + q)/2$:

```
Point p(1,2.2), q(3.4,5.6);
Point h = p.halfway(q);
```

You can write this function directly, or you could write functions `Add` and `Scale` and combine these. (Later you will learn about operator overloading.)

How would you print out a *Point* to make sure you compute the halfway point correctly?

Exercise 18.6. Make a default constructor for the point class:

```
Point() { /* default code */ }
```

which you can use as:

```
Point p;
```

but which gives an indication that it is undefined:

Code:

```
1 Point p3;
2 cout << "Uninitialized point:"
3      << '\n';
4 p3.printout();
5 cout << "Using uninitialized point:"
6      << '\n';
7 auto p4 = Point(4,5)+p3;
8 p4.printout();
```

Output
[geom] linearnan:

```
Uninitialized point:
Point: nan,nan
Using uninitialized point:
Point: nan,nan
```

Hint: see section 15.3.3Not-a-numbersubsection.15.3.3.

Exercise 18.7. Revisit exercise 18.2Basic functionsexcounter.18.2 using the `Point` class. Your code should now look like:

```
newpoint = point.rotate(alpha);
```

Exercise 18.8. Advanced. Can you make a `Point` class that can accommodate any number of space dimensions? Hint: use a `vector`; section **??**. Can you make a constructor where you do not specify the space dimension explicitly?

18.3 Using one class in another

Before doing this section, make sure you study section **??**.

Exercise 18.9. Make a class `LinearFunction` with a constructor:

```
LinearFunction( Point input_p1,Point input_p2 );
```

and a member function

```
float evaluate_at( float x );
```

which you can use as:

```
LinearFunction line(p1,p2);
cout << "Value at 4.0: " << line.evaluate_at(4.0) << endl;
```

Exercise 18.10. Make a class `LinearFunction` with two constructors:

```
LinearFunction( Point input_p2 );
LinearFunction( Point input_p1,Point input_p2 );
```

where the first stands for a line through the origin.
Implement again the `evaluate` function so that

```
LinearFunction line(p1,p2);
cout << "Value at 4.0: " << line.evaluate_at(4.0) << endl;
```

Exercise 18.11. Revisit exercises 18.2Basic functionsexcounter.18.2 and 18.7Point classexcounter.18.7, introducing a `Matrix` class. Your code can now look like

```
newpoint = point.apply(rotation_matrix);
```

or

```
newpoint = rotation_matrix.apply(point);
```

Can you argue in favor of either one?

Suppose you want to write a `Rectangle` class, which could have methods such as `float Rectangle::area()` or `bool Rectangle::contains(Point)`. Since rectangle has four corners, you could store four `Point` objects in each `Rectangle` object. However, there is redundancy there: you only need three points to infer the fourth. Let's consider the case of a rectangle with sides that are horizontal and vertical; then you need only two points.

Intended API:

```
float Rectangle::area();
```

It would be convenient to store width and height; for

```
bool Rectangle::contains(Point);
```

it would be convenient to store bottomleft/topright points.

Exercise 18.12.

1. Make a class `Rectangle` (sides parallel to axes) with a constructor:

   ```
   Rectangle(Point botleft,float width,float height);
   ```

   The logical implementation is to store these quantities. Implement methods:

   ```
   float area(); float rightedge_x(); float topedge_y();
   ```

   and write a main program to test these.
2. Add a second constructor

   ```
   Rectangle(Point botleft,Point topright);
   ```

   Can you figure out how to use *member initializer* lists for the constructors?

Exercise 18.13. Make a copy of your solution of the previous exercise, and redesign your class so that it stores two `Point` objects. Your main program should not change.

The previous exercise illustrates an important point: for well designed classes you can change the implementation (for instance motivated by efficiency) while the program that uses the class does not change.

18.4 Is-a relationship

Before doing this section, make sure you study section **??**.

Exercise 18.14. Take your code where a `Rectangle` was defined from one point, width, and height.

Make a class `Square` that inherits from `Rectangle`. It should have the function `area` defined, inherited from `Rectangle`.

First ask yourself: what should the constructor of a `Square` look like?

Exercise 18.15. Revisit the `LinearFunction` class. Add methods `slope` and `intercept`.

Now generalize `LinearFunction` to `StraightLine` class. These two are almost the same except for vertical lines. The `slope` and `intercept` do not apply to vertical lines, so design `StraightLine` so that it stores the defining points internally. Let `LinearFunction` inherit.

18.5 Pointers

Before doing this section, make sure you study section 11.1.

The following exercise is a little artificial.

Exercise 18.16. Make a `DynRectangle` class, which is constructed from two shared-pointers-to-`Point` objects:

```
auto
  origin  = make_shared<Point>(0,0),
  fivetwo = make_shared<Point>(5,2);
DynRectangle lielow( origin,fivetwo );
```

Calculate the area, scale the top-right point, and recalculate the area:

Code:

```
cout << "Area: " <<  lielow.area() <<
    '\n';
 /* ... */
// scale the 'fivetwo' point by two
cout << "Area: " <<  lielow.area() <<
    '\n';
```

Output

```
[pointer] dynrect:
Area: 10
Area: 40
```

You can base this off the file `pointrectangle.cxx` in the repository

18.6 More stuff

Before doing this section, make sure you study section 10.3.

The `Rectangle` class stores at most one corner, but may be convenient to sometimes have an array of all four corners.

Exercise 18.17. Add a method

```
const vector<Point> &corners()
```

to the `Rectangle` class. The result is an array of all four corners, not in any order. Show by a compiler error that the array can not be altered.

Before doing this section, make sure you study section **??**.

Exercise 18.18. Revisit exercise 18.5 and replace the `add` and `scale` functions by overloaded operators.

Hint: for the `add` function you may need '`this`'.

Chapter 19

Zero finding

19.1 Root finding by bisection

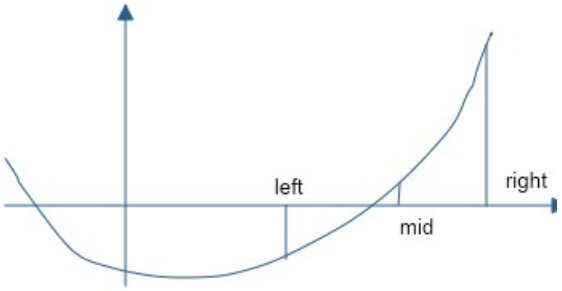


Figure 19.1: Root finding by interval bisection

For many functions f, finding their zeros, that is, the values x for which $f(x) = 0$, can not be done analytically. You then have to resort to numerical *root finding* schemes. In this project you will develop gradually more complicated implementations of a simple scheme: root finding by *bisection*.

In this scheme, you start with two points where the function has opposite signs, and move either the left or right point to the mid point, depending on what sign the function has there. See figure 19.1.

In section 19.2 we will then look at Newton's method.

Here we will not be interested in mathematical differences between the methods, though these are important: we will use these methods to exercise some programming techniques.

19.1.1 Simple implementation

Before doing this section, make sure you study section **??**.

Before doing this section, make sure you study section **??**.

Let's develop a first implementation step by step. To ensure correctness of our code we will use a Test-Driven Development (TDD) approach: for each bit of functionality we write a test to ensure its correctness before we integrate it in the larger code. (For more about TDD, and in particular the Catch2 framework, see section 36.2Unit testing frameworkssection.36.2.)

19.1.2 Polynomials

First of all, we need to have a way to represent polynomials. For a polynomial of degree d we need $d+1$ coefficients:

$$f(x) = c_0 x^d + \cdots + c_{d-1} x^1 + c_d \tag{19.1}$$

We implement this by storing the coefficients in a `vector<double>`. We make the following arbitrary decisions

1. let the first element of this vector be the coefficient of the highest power, and
2. for the coefficients to properly define a polynomial, this leading coefficient has to be nonzero.

Let's start by having a fixed test polynomial, provided by a function `set_coefficients`. For this function to provide a proper polynomial, it has to satisfy the following test:

```
TEST_CASE( "coefficients represent polynomial" "[1]") {
  vector<double> coefficients = { 1.5, 0., -3 };
  REQUIRE( coefficients.size()>0 );
  REQUIRE( coefficients.front()!=0. );
}
```

Exercise 19.1. Write a routine `set_coefficients` that constructs a vector of coefficients:

```
vector<double> coefficients = set_coefficients();
```

and make it satisfy the above conditions.

At first write a hard-coded set of coefficients, then try reading them from the command line.

Exercise 19.2. Bonus: use the `cxxopts` library (section 35.2.2The `cxxopts` librarysubsection.35.2.2) to specify the coefficients on the commandline.

Above we postulated two conditions that an array of numbers should satisfy to qualify as the coefficients of a polynomial. Your code will probably be testing for this, so let's introduce a boolean function `is_proper_polynomial`:

- This function returns `true` if the array of numbers satisfies the two conditions;
- it returns `false` if either condition is not satisfied.

In order to test your function `is_proper_polynomial` you should check that

- it recognizes correct polynomials, and
- it fails for improper coefficients that do not properly define a polynomial.

Exercise 19.3. Write a function `is_proper_polynomial` as described, and write unit tests for it, both passing and failing:

```
vector<double> good = /* proper coefficients */ ;
REQUIRE( is_proper_polynomial(good) );
vector<double> notso = /* improper coefficients */ ;
REQUIRE( not is_proper_polynomial(notso) );
```

Next we need polynomial evaluation. We will build a function `evaluate_at` with the following definition:

```
double evaluate_at( const std::vector<double>& coefficients,double x);
```

You can interpret the array of coefficients in (at least) two ways, but with equation (19.1) we proscribed one particular interpretation.

So we need a test that the coefficients are indeed interpreted with the leading coefficient first, and not with the leading coefficient last. For instance:

```
polynomial second( {2,0,1} );
// correct interpretation: 2x^2 + 1
REQUIRE( second.is_proper() );
REQUIRE( second.evaluate_at(2) == Catch::Approx(9) );
// wrong interpretation: 1x^2 + 2
REQUIRE( second.evaluate_at(2) != Catch::Approx(6) );
```

(where we have left out the `TEST_CASE` header.)

Now we write the function that passes these tests:

Exercise 19.4. Write a function `evaluate_at` which computes

$$y \leftarrow f(x).$$

and confirm that it passes the above tests.

```
double evaluate_at( polynomial coefficients,double x);
```

For bonus points, look up *Horner's rule* and implement it.

With the polynomial function implemented, we can start working towards the algorithm.

19.1.3 Left/right search points

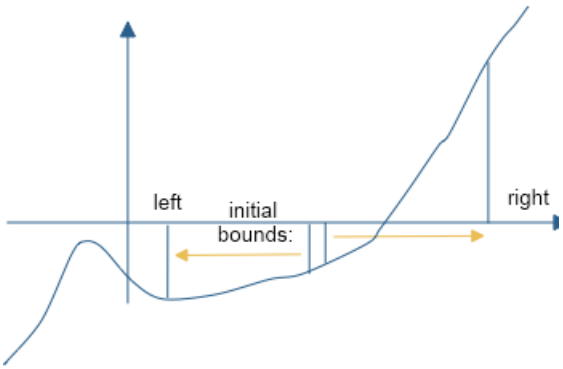


Figure 19.2: Setting the initial search points

Suppose x_-, x_+ are such that

$$x_- < x_+, \quad \text{and} \quad f(x_-) \cdot f(x_+) < 0,$$

that is, the function values in the left and right point are of opposite sign. Then there is a zero in the interval (x_-, x_+); see figure 19.1.

But how to find these outer bounds on the search?

If the polynomial is of odd degree you can find x_-, x_+ by going far enough to the left and right from any two starting points. For even degree there is no such simple algorithm (indeed, there may not be a zero) so we abandon the attempt.

We start by writing a function `is_odd` that tests whether the polynomial is of odd degree.

Exercise 19.5. Make the following code work:

```
if ( not is_odd(coefficients) ) {
  cout << "This program only works for odd-degree polynomials\n";
  exit(1);
}
```

You could test the above as:

```
polynomial second{2,0,1}; // 2x^2 + 1
REQUIRE( not is_odd(second) );
polynomial third{3,2,0,1}; // 3x^3 + 2x^2 + 1
REQUIRE( is_odd(third) );
```

Now we can find x_-, x_+: start with some interval and move the end points out until the function values have opposite sign.

Exercise 19.6. Write a function `find_initial_bounds` which computes x_-, x_+ such that

$$f(x_-) < 0 < f(x_+) \quad \text{or} \quad f(x_+) < 0 < f(x_-)$$

Address the following concerns:

1. What is a good prototype for the function?
2. How do move the points far enough out to satisfy this condition?
3. Can you compute the above test more compactly?

Since finding a left and right point with a zero in between is not always possible for polynomials of even degree, we completely reject this case. In the following test we throw an exception (see section 13.2.2Exception handlingsubsection.13.2.2, in particularly 13.2.2.3Throw your own exceptionssubsubsection.13.2.2.3) for polynomials of even degree:

```
right = left+1;
polynomial second{2,0,1}; // 2x^2 + 1
REQUIRE_THROWS( find_initial_bounds(second,left,right) );
polynomial third{3,2,0,1}; // 3x^3 + 2x^2 + 1
REQUIRE_NOTHROW( find_initial_bounds(third,left,right) );
REQUIRE( left<right );
```

Make sure your code passes these tests. What test do you need to add for the function values?

19.1.4 Root finding

The root finding process globally looks as follows:

- You start with points x_-, x_+ where the function has opposite sign; then you know that there is a zero between them.
- The bisection method for finding this zero looks at the halfway point, and based on the function value in the mid point:
- moves one of the bounds to the mid point, such that the function again has opposite signs in the left and right search point.

The structure of the code is as follows:

```
double find_zero( /* something */ ) {
  while ( /* left and right too far apart */ ) {
    // move bounds left and right closer together
  }
  return something;
}
```

Again, we test all the functionality separately. In this case this means that moving the bounds should be a testable step.

Exercise 19.7. Write a function `move_bounds_closer` and test it.

```
void move_bounds_closer
    ( std::vector<double> coefficients,
      double& left,double& right );
```

Implement some unit tests on this function.

Finally, we put everything together in the top level function `find_zero`.

Exercise 19.8. Make this call work:

```
auto zero = find_zero( coefficients, 1.e-8 );
cout << "Found root " << zero
     << " with value " << evaluate_at(coefficients,zero) << '\n';
```

Design unit tests, including on the precision attained, and make sure your code passes them.

19.1.5 Object implementation

Revisit the exercises of section 19.1.1 and introduce a `polynomial` class that stores the polynomial coefficients. Several functions now become members of this class.

Also update the unit tests.

Some further suggestions:

1. Can you make your polynomial class look like a function?

```
class Polynomial {
  /* ... */
}
main () {
  Polynomial p;
  float y = p(x);
}
```

See section **??**.

2. Can you generalize the polynomial class, for instance to the case of special forms such as $(1+x)^n$?
3. Templatize your polynomials: see next subsection.

19.1.6 Templating

In the implementations so far we used `double` for the numerical type. Make a templated version that works both with `float` and `double`.

Can you see a difference in attainable precision between the two types?

19.2 Newton's method

Before doing this section, make sure you study section 8.

In this section we look at *Newton's method*. This is an iterative method for finding zeros of a function f, that is, it computes a sequence of values $\{x_n\}_n$, so that $f(x_n) \to 0$. The sequence is defined by

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

with x_0 arbitrarily chosen. For details, see HPC book [11], section 22.

While in practice Newton's method is used for complicated functions, here we will look at a simple example, which actually has a basis in computing history. Early computers had no hardware for computing a square root. Instead, they used Newton's method.

Suppose you have a positive value y and you want want to compute $x = \sqrt{y}$. 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 $f_y(x) \approx 0$.

We will not go into the matter of how to choose a sophisticated stopping test for the iteration; that is a matter for a numerical analysis course, not a programming course.

19.2.1 Function implementation

It is of course simple to code this specific case; it should take you about 10 lines. However, we want to have a general code that takes any two functions f, f', and then uses Newton's method to find a zero of our specific function f.

Exercise 19.9.

- 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 $\sqrt{y}$.

19.2.2 Using lambdas

Above you wrote functions conforming to:

```
double f(double x);
double fprime(double x);
```

and the algorithm:

```
double x{1.};
while ( true ) {
  auto fx = f(x);
  cout << "f( " << x << " ) = " << fx << '\n';
  if (std::abs(fx)<1.e-10 ) break;
  x = x - fx/fprime(x);
}
```

Exercise 19.10. Rewrite your code to use lambda functions for `f` and `fprime`.

You can base this off the file `newton.cxx` in the repository

Next, we make the code modular by writing a general function `newton_root`, that contains the Newton method of the previous exercise. Since it has to work for any functions f, f', you have to pass the objective function and the derivative as arguments:

```
double root = newton_root( f,fprime );
```

Exercise 19.11. Rewrite the Newton exercise above to use a function that is used as:

```
double root = newton_root( f,fprime );
```

Call the function

1. first with the lambda variables you already created;
2. but in a better variant, directly with the lambda expressions as arguments, that is, without assigning them to variables.

Next we extend functionality, but not by changing the root finding function: instead, we use a more general way of specifying the objective function and derivative.

Exercise 19.12. Extend the newton exercise to compute roots in a loop:

```
for (int n=2; n<=8; ++n) {
  cout << "sqrt(" << n << ") = "
       << newton_root(
  /* ... */
                     )
```

```
            << '\n';
```

Without lambdas, you would define a function

```
double squared_minus_n( double x,int n ) {
  return x*x-n; }
```

However, the `newton_root` function takes a function of only a real argument. Use a capture to make `f` dependent on the integer parameter.

Exercise 19.13. You don't need the gradient as an explicit function: you can approximate it as

$$f'(x) = \big(f(x+h) - f(x)\big)/h$$

for some value of h.

Write a version of the root finding function that only takes the objective function:

```
double newton_root( function< double(double)> f )
```

You can use a fixed value `h=1e-6`.

Do not reimplement the whole newton method: instead create a lambda for the gradient and pass it to the function `newton_root` you coded earlier.

Exercise 19.14. Bonus: can you compute logarithms through Newton's method?

19.2.3 Templated implementation

Newton's method works equally well for complex numbers as for real numbers.

Exercise 19.15. Rewrite your Newton program so that it works for complex numbers:

```
complex<double> z{.5,.5};
while ( true ) {
  auto fz = f(z);
  cout << "f( " << z << " ) = " << fz << '\n';
  if (std::abs(fz)<1.e-10 ) break;
  z = z - fz/fprime(z);
}
```

You may run into the problem that you can not operate immediately between a complex number and a `float` or `double`. Use `static_cast`; see section 15.2.1.

So do you have to write two separate implementations, one for reals, and one for complex numbers? (And maybe you need two separate ones for `float` and `double`!)

This is where *templates* come in handy; chapter 12Templateschapter.12.

You can templatize your Newton function and derivative:

```
template<typename T>
T f(T x) { return x*x - 2; };
```

```
template<typename T>
T fprime(T x) { return 2 * x; };
```

and then write

```
double x{1.};
while ( true ) {
  auto fx = f<double>(x);
  cout << "f( " << x << " ) = " << fx << '\n';
  if (std::abs(fx)<1.e-10 ) break;
  x = x - fx/fprime<double>(x);
}
```

Exercise 19.16. Update your Newton program with templates. If you have it working for `double`, try using `complex<double>`. Does it work?

Exercise 19.17. Use your complex Newton method to compute $\sqrt{2}$. Does it work?

How about $\sqrt{-2}$?

Exercise 19.18. Can you templatize your Newton code that used lambda expressions? Your function header would now be:

```
template<typename T>
T newton_root
   ( function< T(T) > f,
     function< T(T) > fprime,
     T init) {
```

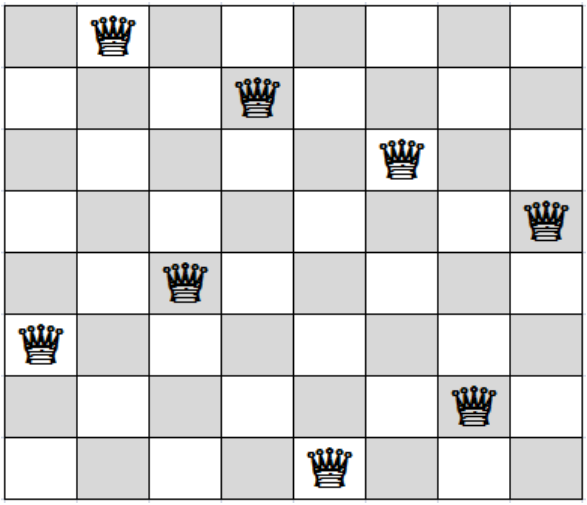
You would for instance compute $\sqrt{2}$ as:

```
cout << "sqrt -2 = " <<
     newton_root<complex<double>>
          ( [] (complex<double> x) {
               return x*x + static_cast<complex<double>>(2); },
            [] (complex<double> x) {
              return x * static_cast<complex<double>>(2); },
            complex<double>{.1,.1}
            )
     << '\n';
```

Chapter 20

Eight queens

A famous exercise for recursive programming is the *eight queens* problem: Is it possible to position eight queens on a chess board, so that no two queens 'threaten' each other, according to the rules of chess?



20.1 Problem statement

The precise statement of the 'eight queens problem' is:

- Put eight pieces on an 8×8 board, no two pieces on the same square; so that
- no two pieces are on the same row,
- no two pieces are on the same column, and
- no two pieces are on the same diagonal.

A systematic solution would run:

1. put a piece anywhere in the first row;
2. for each choice in the first row, try all positions in the second row;
3. for all choices in the first two rows, try all positions in the third row;
4. when you have a piece in all eight rows, evaluate the board to see if it satisfies the condition.

Exercise 20.1. This algorithm will generate all 8^8 boards. Do you see at least one way to speed up the search?

Since the number eight is, for now, fixed, you could write this code as an eight-deep loop nest. However that is not elegant. For example, the only reason for the number 8 in the above exposition is that this is the traditional size of a chess board. The problem, stated more abstractly as placing n queens on an $n \times n$ board, has solutions for $n \geq 4$.

20.2 Solving the eight queens problem, basic approach

This problem requires you to know about arrays/vectors; chapter **??**. Also, see chapter 36Unit testing and Test-Driven Developmentchapter.36 for TDD. Finally, see section **??** for `std::optional`.

The basic strategy will be that we fill consecutive rows, by indicating each time which column will be occupied by the next queen. Using an Object-Oriented (OO) strategy, we make a class `ChessBoard`, that holds a partially filled board.

The basic solution strategy is recursive:

- Let `current` be the current, partially filled board;
- We call `current.place_queens()` that tries to finish the board;
- However, with recursion, this method only fills one row, and then calls `place_queens` on this new board.

So

```
ChessBoard::place_queens() {
  // for c = 1 ... number of columns:
  //   make a copy of the board
  //   put a queen in the next row, column c, of the copy
  //   and call place_queens() on that copy;
  //   investigate the result.....
}
```

This routine returns either a solution, or an indication that no solution was possible.

In the next section we will develop a solution systematically in a TDD manner.

20.3 Developing a solution by TDD

We now gradually develop the OO solution to the eight queens problem, using test-driven development.

The board We start by constructing a board, with a constructor that only indicates the size of the problem:

```
ChessBoard(int n);
```

This is a 'generalized chess board' of size $n \times n$, and initially it should be empty.

Exercise 20.2. Write this constructor, for an empty board of size $n \times n$.

Note that the implementation of the board is totally up to you. In the following you will get tests for functionality that you need to satisfy, but any implementation that makes this true is a correct solution.

Bookkeeping: what's the next row? Assuming that we fill in the board row-by-row, we have an auxiliary function that returns the next row to be filled:

```
int next_row_to_be_filled()
```

This gives us our first simple test: on an empty board, the row to be filled in is row zero.

Exercise 20.3. Write this method and make sure that it passes the test for an empty board.

```
TEST_CASE( "empty board","[1]" ) {
  constexpr int n=10;
  ChessBoard empty(n);
  REQUIRE( empty.next_row_to_be_filled()==0 );
}
```

By the rules of TDD you can actually write the method so that it only satisfies the test for the empty board. Later, we will test that this method gives the right result after we have filled in a couple of rows, and then of course your implementation needs to be general.

Place one queen Next, we have a function to place the next queen, whether this gives a feasible board (meaning that no pieces can capture each other) or not:

```
void place_next_queen_at_column(int i);
```

This method should first of all catch incorrect indexing: we assume that the placement routine throws an exception for invalid column numbers.

```
ChessBoard::place_next_queen_at_column( int c ) {
  if ( /* c is outside the board */ )
    throw(1); // or some other exception.
```

(Suppose you didn't test for incorrect indexing. Can you construct a simple 'cheating' solution at any size?)

Exercise 20.4. Write this method, and make sure it passes the following test for valid and invalid column numbers:

```
REQUIRE_THROWS( empty.place_next_queen_at_column(-1) );
REQUIRE_THROWS( empty.place_next_queen_at_column(n) );
REQUIRE_NOTHROW( empty.place_next_queen_at_column(0) );
REQUIRE( empty.next_row_to_be_filled()==1 );
```

(From now on we'll only give the body of the test.)

Now it's time to start writing some serious stuff.

Is a (partial) board feasible? If you have a board, even partial, you want to test if it's feasible, meaning that the queens that have been placed can not capture each other.

The prototype of this method is:

```
bool feasible()
```

This test has to work for simple cases to begin with: an empty board is feasible, as is a board with only one piece.

```
ChessBoard empty(n);
REQUIRE( empty.feasible() );
```

```
ChessBoard one = empty;
one.place_next_queen_at_column(0);
REQUIRE( one.next_row_to_be_filled()==1 );
REQUIRE( one.feasible() );
```

Exercise 20.5. Write the method and make sure it passes these tests.

We shouldn't only do successful tests, sometimes referred to as the 'happy path' through the code. For instance, if we put two queens in the same column, the test should fail.

Exercise 20.6. Take the above initial attempt with a queen in position $(0, 0)$, and add another queen in column zero of the next row. Check that it passes the test:

```
ChessBoard collide = one;
// place a queen in a 'colliding' location
collide.place_next_queen_at_column(0);
// and test that this is not feasible
REQUIRE( not collide.feasible() );
```

Add a few tests more of your own. (These will not be exercised by the submission script, but you may find them useful anyway.)

Testing configurations If we want to test the feasibility of non-trivial configurations, it is a good idea to be able to 'create' solutions. For this we need a second type of constructor where we construct a fully filled chess board from the locations of the pieces.

```
ChessBoard( int n,vector<int> cols );
ChessBoard( vector<int> cols );
```

- If the constructor is called with only a vector, this describes a full board.
- Adding an integer parameter indicates the size of the board, and the vector describes only the rows that have been filled in.

Exercise 20.7. Write these constructors, and test that an explicitly given solution is a feasible board:

```
ChessBoard five( {0,3,1,4,2} );
REQUIRE( five.feasible() );
```

For an elegant approach to implementing this, see *delegating constructors*; section **??**.

Ultimately we have to write the tricky stuff.

20.4 The recursive solution method

The main function

```
optional<ChessBoard> place_queens()
```

takes a board, empty or not, and tries to fill the remaining rows.

One problem is that this method needs to be able to communicate that, given some initial configuration, no solution is possible. For this, we let the return type of `place_queens` be `optional<ChessBoard>`:

- if it is possible to finish the current board resulting in a solution, we return that filled board;
- otherwise we return `{}`, indicating that no solution was possible.

With the recursive strategy discussed in section 20.2, this placement method has roughly the following structure:

```
place_queens() {
  for ( int col=0; col<n; col++ ) {
    ChessBoard next = *this;
    // put a queen in column col on the 'next' board
    // if this is feasible and full, we have a solution
    // if it is feasible but no full, recurse
  }
}
```

The line

```
ChessBoard next = *this;
```

makes a copy of the object you're in.

Remark 13 *Another approach would be to make a recursive function*

```
bool place_queen( const ChessBoard& current, ChessBoard &next );
// true if possible, false is not
```

The final step Above you coded the method `feasible` that tested whether a board is still a candidate for a solution. Since this routine works for any partially filled board, you also need a method to test if you're done.

Exercise 20.8. Write a method

```
bool filled();
```

and write a test for it, both positive and negative.

Now that you can recognize solutions, it's time to write the solution routine.

Exercise 20.9. Write the method

```
optional<ChessBoard> place_queens()
```

Because the function `place_queens` is recursive, it is a little hard to test in its entirety.

We start with a simpler test: if you almost have the solution, it can do the last step.

Exercise 20.10. Use the constructor

```
ChessBoard( int n,vector<int> cols )
```

to generate a board that has all but the last row filled in, and that is still feasible. Test that you can find the solution:

```
ChessBoard almost( 4, {1,3,0} );
```

```
auto solution = almost.place_queens();
REQUIRE( solution.has_value() );
REQUIRE( solution->filled() );
```

Since this test only fills in the last row, it only does one loop, so printing out diagnostics is possible, without getting overwhelmed in tons of output.

Solutions and non-solutions Now that you have the solution routine, test that it works starting from an empty board. For instance, confirm there are no 3×3 solutions:

```
TEST_CASE( "no 3x3 solutions","[9]" ) {
  ChessBoard three(3);
  auto solution = three.place_queens();
  REQUIRE( not solution.has_value() );
}
```

On the other hand, 4×4 solutions do exist:

```
TEST_CASE( "there are 4x4 solutions","[10]" ) {
  ChessBoard four(4);
  auto solution = four.place_queens();
  REQUIRE( solution.has_value() );
}
```

Exercise 20.11. (Optional) Can you modify your code so that it counts all the possible solutions?

Exercise 20.12. (Optional) How does the time to solution behave as function of n?

Chapter 21

Infectious disease simulation

This section contains a sequence of exercises that builds up to a somewhat realistic simulation of the spread of infectious diseases.

21.1 Model design

It is possible to model disease propagation statistically, but here we will build an explicit simulation: we will maintain an explicit description of all the people in the population, and track for each of them their status.

We will use a simple model where a person can be:

- sick: when they are sick, they can infect other people;
- susceptible: they are healthy, but can be infected;
- recovered: they have been sick, but no longer carry the disease, and can not be infected for a second time;
- vaccinated: they are healthy, do not carry the disease, and can not be infected.

In more complicated models a person could be infectious during only part of their illness, or there could be secondary infections with other diseases, et cetera. We keep it simple here: any sick person can infect others while they are sick.

In the exercises below we will gradually develop a model of how the disease spreads from an initial source of infection. The program will then track the population from day to day, running indefinitely until none of the population is sick. Since there is no re-infection, the run will always end. Later we will add mutation to the model which can extend the duration of the epidemic.

21.1.1 Other ways of modeling

Instead of capturing every single person in code, a 'contact network' model, it is possible to use an Ordinary Diffential Equation (ODE) approach to disease modeling. You would then model the percentage of infected persons by a single scalar, and derive relations for that and other scalars [3, 16].

This is known as a 'compartmental model', where each of the three SIR states, Susceptible, Infected, Removed, is a compartment: a section of the population. Both the contact network and the compartmental model capture part of the truth. In fact, they can be combined. We can consider a country as a set of cities,

where people travel between any pair of cities. We then use a compartmental model inside a city, and a contact network between cities.

In this project we will only use the network model.

21.2 Coding

The following sections are a step-by-step development of the code for this project. Later we will discuss running this code as a scientific experiment.

Remark 14 *In various places you need a random number generator. You can use the C language random number generator (section* **??***), or the new STL one in section* **??***.*

21.2.1 Person basics

The most basic component of a disease simulation is to infect a person with a disease, and see the time development of that infection. Thus you need a person class and a disease class.

Before you start coding, ask yourself what behaviors these classes need to support.

- Person. The basic methods for a person are
  1. Get infected;
  2. Get vaccinated; and
  3. Progress by one day.

  Furthermore you may want to query what the state of the person is: are they healthy, sick, recovered?
- Disease. For now, a disease itself doesn't do much. (Later in the project you may want it to have a method `mutate`.) However, you may want to query certain properties:
  1. Chance of transmission; and
  2. Number of days a person stays sick when infected.

A test for a single person could have output along the following lines:

```
On day 10, Joe is susceptible
On day 11, Joe is susceptible
On day 12, Joe is susceptible
On day 13, Joe is susceptible
On day 14, Joe is sick (5 days to go)
On day 15, Joe is sick (4 days to go)
On day 16, Joe is sick (3 days to go)
On day 17, Joe is sick (2 days to go)
On day 18, Joe is sick (1 days to go)
On day 19, Joe is recovered
```

Exercise 21.1. Write a `Person` class with methods:

- `status_string()` : returns a description of the person's state as a `string`;
- `one_more_day()` : update the person's status to the next day;
- `infect(s)` : infect a person with a disease, where the disease object

```
Disease s(n);
```

is specified to run for n days.

Your main program could for instance look like:

```
for ( int step = 1; ; ++step ) {

  joe.one_more_day();
  /* ... */
  cout << "On day " << step << ", Joe is "
       << joe.status_string() << '\n';
  if (joe.is_recovered())
    break;
}
```

where the infection part has been left out.

Your main concern is how to model the internal state of a person. This is actually two separate issues:

1. the state, and
2. if sick, how many days to go to recover.

You can find a way to implement this with a single integer, but it's better to use two. Also, write enough support methods such as the `is_recovered` test.

21.2.1.1 Person tests

It is easy to write code that seems to be the right thing, but does not behave correctly in all cases. So it is a good idea to subject your code to some systematic tests.

Make sure your `Person` objects pass these tests:

- After being infected with a 100% transmittable disease, they should register as sick.
- If they are vaccinated or recovered, and they come in contact with such a disease, they stay in their original state.
- If a disease has a transmission chance of 50%, and a number of people come into contact with it, about half of them should get sick. This test maybe a little tricky to write.

Can you use the *Catch2* unittesting framework? See section 35.3Catch2 unit testingsection.35.3.

21.2.2 Interaction

Next we model interactions between people: one person is healthy, another is infected, and when the two come into contact the disease may be transferred.

```
Person infected,healthy;
infected.infect(flu);
/* ... */
  healthy.touch(infected);
```

The disease has a certain probability of being transferred, so you need to specify that probability. You could let the declaration be:

```
Disease flu( 5, 0.3 );
```

where the first parameter is the number of days an infection lasts, and the second the transfer probability.

Exercise 21.2. Add a transmission probability to the `Disease` class, and add a `touch` method to the `Person` class. Design and run some tests.

Exercise 21.3. Bonus: can you get the following disease specification to work?

```
Disease flu;
flu.duration() = 20;
flu.transfer_probability() = p;
```

Why could you consider this better than the earlier suggested syntax?

21.2.2.1 Interaction tests

Adapt the above tests, but now a person comes in contact with an infected person, rather than directly with a disease.

21.2.3 Population

Next we need a `Population` class, where a population contains a `vector` consisting of `Person` objects. Initially we only infect one person, and there is no transmission of the disease.

The `Population` class should at least have the following methods:

- `random_infection` to start out with an infected segment of the population;
- `random_vaccination` to start out with a number of vaccinated individuals.
- counting functions `count_infected` and `count_vaccinated`.

To run a realistic simulation you also need a `one_more_day` method that takes the population through a day. This is the heart of your code, and we will develop this gradually in the next section.

21.2.3.1 Population tests

Most population testing will be done in the following section. For now, make sure you pass the following tests:

- With a vaccination percentage of 100%, everyone should indeed be vaccinated.

21.3 Epidemic simulation

To simulate the spread of a disease through the population, we need an update method that progresses the population through one day:

- Sick people come into contact with a number of other members of the populace;
- and everyone gets one day older, meaning mostly that sick people get one day closer to recovery.

We develop this in a couple of steps.

21.3.1 No contact

At first assume that people have no contact, so the disease ends with the people it starts with.

The trace output should look something like:

```
Size of
population?
In step   1 #sick:    1 :  ? ? ? ? ? ? ? ? ? + ? ? ? ? ? ? ? ? ?
In step   2 #sick:    1 :  ? ? ? ? ? ? ? ? ? + ? ? ? ? ? ? ? ? ?
In step   3 #sick:    1 :  ? ? ? ? ? ? ? ? ? + ? ? ? ? ? ? ? ? ?
In step   4 #sick:    1 :  ? ? ? ? ? ? ? ? ? + ? ? ? ? ? ? ? ? ?
In step   5 #sick:    1 :  ? ? ? ? ? ? ? ? ? + ? ? ? ? ? ? ? ? ?
In step   6 #sick:    0 :  ? ? ? ? ? ? ? ? ? - ? ? ? ? ? ? ? ? ?
Disease ran its course by step 6
```

Remark 15 *Such a display is good for a sanity check on your program behavior. If you include such displays in your writeup, make sure to use a monospace font, and don't use a population size that needs line wrapping. In further testing, you should use large populations, but do not include these displays.*

Exercise 21.4. Program a population without infection.

- Write the `Population` class. The constructor takes the number of people:

  ```
  Population population(npeople);
  ```

- Write a method that infects a number of random people:

  ```
  population.random_infection(fever,initial_infect);
  ```

- Write a method `count_infected` that counts how many people are infected.
- Write an `one_more_day` method that updates all persons in the population.
- Loop the `one_more_day` method until no people are infected: the `Population::one_more_day` method should apply `Person::one_more_day` to all person in the population.

Write a routine that displays the state of the popular, using for instance: `?` for susceptible, + for infected, − for recovered.

21.3.1.1 Tests

Test that for the duration of the disease, the number of infected people stays constant, and that the sum of healthy and infected people stays equal to the population size.

21.3.2 Contagion

This past exercise was too simplistic: the original patient zero was the only one who ever got sick. Now let's incorporate contagion, and investigate the spread of the disease from a single infected person.

We start with a very simple model of infection.

Exercise 21.5. Write a simulation where in each step the direct neighbors of an infected person can now get sick themselves.

Run a number of simulations with population sizes and contagion probabilities. Are there cases where people escape getting sick?

21.3.2.1 Tests

Do some sanity tests:

- If one person is infected with a disease with $p = 1$, the next day there should be 3 people sick. Unless the infected person is the first or last: then there are two.
- If person 0 is infected, and $p = 1$, the simulation should run for a number of days equal to the size of the population.
- How is the previous case if $p = 0.5$?

21.3.3 Vaccination

Exercise 21.6. Incorporate vaccination: read another number representing the percentage of people that has been vaccinated. Choose those members of the population randomly.

Describe the effect of vaccinated people on the spread of the disease. Why is this model unrealistic?

21.3.4 Spreading

To make the simulation more realistic, we let every sick person come into contact with a fixed number of random people every day. This gives us more or less the *SIR model*; https://en.wikipedia.org/wiki/Epidemic_model.

Set the number of people that a person comes into contact with, per day, to 6 or so. (You can also let this be an upper bound for a random value, but that does not essentially change the simulation.) You have already programmed the probability that a person who comes in contact with an infected person gets sick themselves. Again start the simulation with a single infected person.

Exercise 21.7. Code the random interactions. Now run a number of simulations varying

- The percentage of people vaccinated, and
- the chance the disease is transmitted on contact.

Record how long the disease runs through the population. With a fixed number of contacts and probability of transmission, how is this number of function of the percentage that is vaccinated?

Report this function as a table or graph. Make sure you have enough data points for a meaningful conclusion. Use a realistic population size. You can also do multiple runs and report the average, to even out the effect of the random number generator.

Exercise 21.8. Investigate the matter of 'herd immunity': if enough people are vaccinated, then some people who are not vaccinated will still never get sick. Let's say you want to have the probability of being not vaccinated, yet never getting sick, to be over 95 percent. Investigate the percentage of vaccination that is needed for this as a function of the contagiousness of the disease.

As in the previous exercise, make sure your data set is large enough.

Remark 16 *The screen output you used above is good for sanity checks on small problems. However, for realistic simulations you have to think what is a realistic population size. If your university campus is a population where random people are likely to meet each other, what would be a population size to model that? How about the city where you live?*

Likewise, if you test different vaccination rates, what granularity do you use? With increases of 5 or 10 percent you can print all results to you screen, but you may miss things. Don't be afraid to generate large amount of data and feed them directly to a graphing program.

21.3.5 Mutation

The Covid years have shown how important mutations of an original virus can be. Next, you can include mutation in your project. We model this as follows:

- Every so many transmissions, a virus will mutate into a new variant.
- A person who has recovered from one variant is still susceptible to other variants.
- For simplicity assume that each variant leaves a person sick the same number of days, and
- Vaccination is all-or-nothing: one vaccine is enough to protect against all variant;
- On the other hand, having recovered from one variant is not protection against others.

Implementation-wise speaking, we model this as follows. First of all, we need a `Disease` class, so that we can infect a person with an explicit virus;

```
void touch( const Person&,long int ps=0 );
void infect( const Disease& );
```

A `Disease` object now carries the information such as the chance of transmission, or how a long a person stays under the weather. Modeling mutation is a little tricky. You could do it as follows:

- There is a global `variants` counter for new virus variants, and a global `transmissions` counter.
- Every time a person infects another, the newly infected person gets a new `Disease` object, with the current variant, and the transmissions counter is updated.
- There is a parameter that determines after how many transmissions the disease mutates. If there is a mutation, the global `variants` counter is updated, and from that point on, every infection is with the new variant. (Note: this is not very realistic. You are free to come up with a better model.)
- A each `Person` object has a vector of variants that they are recovered from; recovery from one variant only makes them immune from that specific variant, not from others.

Exercise 21.9. Add mutation to your model. Experiment with the mutation rate: as the mutation rate increases, the disease should stay in the population longer. Does the relation with vaccination rate change that you observed before?

21.3.6 Diseases without vaccine: Ebola and Covid-19

This section is optional, for bonus points

The project so far applies to diseases for which a vaccine is available, such as MMR for measles, mumps and rubella. The analysis becomes different if no vaccine exists, such as is the case for *Ebola* and *Covid-19*, as of this writing.

Instead, you need to incorporate 'social distancing' into your code: people do not get in touch with random others anymore, but only those in a very limited social circle. Design a model distance function, and explore various settings.

The difference between Ebola and Covid-19 is how long an infection can go unnoticed: the *incubation period*. With Ebola, infection is almost immediately apparent, so such people are removed from the general population and treated in a hospital. For Covid-19, a person can be infected, and infect others, for a number of days before they are sequestered from the population.

Add this parameter to your simulation and explore the behavior of the disease as a function of it.

21.4 Ethics

The subject of infectious diseases and vaccination is full of ethical questions. The main one is *The chances of something happening to me are very small, so why shouldn't I bend the rules a little?*. This reasoning is most often applied to vaccination, where people for some reason or other refuse to get vaccinated.

Explore this question and others you may come up with: it is clear that everyone bending the rules will have disastrous consequences, but what if only a few people did this?

21.5 Project writeup and submission

21.5.1 Program files

In the course of this project you have written more than one main program, but some code is shared between the multiple programs. Organize your code with one file for each main program, and a single 'library' file with the class methods.

You can do this two ways:

1. You make a 'library' file, say `infect_lib.cc`, and your main programs each have a line

   ```
   #include "infect_lib.cc"
   ```

   This is not the best solution, but it is acceptable for now.
2. The better solution requires you to use *separate compilation* for building the program, and you need a *header* file. You would now have `infect_lib.cc` which is compiled separately, and `infect_lib.h` which is included both in the library file and the main program:

   ```
   #include "infect_lib.h"
   ```

   See section **??** for more information.

Submit all source files with instructions on how to build all the main programs. You can put these instructions in a file with a descriptive name such as `README` or `INSTALL`, or you can use a *makefile*.

21.5.2 Writeup

In the writeup, describe the 'experiments' you have performed and the conclusions you draw from them. The exercises above give you a number of questions to address.

For each main program, include some sample output, but note that this is no substitute for writing out your conclusions in full sentences.

The exercises in section 21.3.4Spreadingsubsection.21.3.4 ask you to explore the program behavior as a function of one or more parameters. Include a table to report on the behavior you found. You can use Matlab or Matplotlib in Python (or even Excell) to plot your data, but that is not required.

21.6 Bonus: mathematical analysis

The SIR model can also be modeled through coupled difference or differential equations.

1. The number S_i of susceptible people at time i decreases by a fraction

   $$S_{i+1} = S_i(1 - \lambda_i\, dt)$$

   where λ_i is the product of the number of infected people and a constant that reflects the number of meetings and the infectiousness of the disease. We write:

   $$S_{i+1} = S_i(1 - \lambda I_i\, dt)$$

2. The number of infected people similarly increases by $\lambda S_i I_i$, but it also decreases by people recovering (or dying):

   $$I_{i+1} = I_i(1 + \lambda S_i\, dt - \gamma\, dt).$$

3. Finally, the number of 'removed' people equals that last term:

   $$R_{i+1} = R_i(1 + \gamma I_i).$$

Exercise 21.10. Code this scheme. What is the effect of varying dt?

Exercise 21.11. For the disease to become an epidemic, the number of newly infected has to be larger than the number of recovered. That is,

$$\lambda S_i I_i - \gamma I_i > 0 \Leftrightarrow S_i > \gamma/\lambda.$$

Can you observe this in your simulations?

The parameter γ has a simple interpretation. Suppose that a person stays ill for δ days before recovering. If I_t is relatively stable, that means every day the same number of people get infected as recover, and therefore a $1/\delta$ fraction of people recover each day. Thus, γ is the reciprocal of the duration of the infection in a given person.

Chapter 22

Google PageRank

22.1 Basic ideas

We are going to simulate the Internet. In particular, we are going to simulate the *Pagerank* algorithm by which *Google* determines the importance of web pages.

Let's start with some basic classes:

- A `Page` contains some information such as its title and a global numbering in Google's data-center. It also contains a collection of links.
- We represent a link with a pointer to a `Page`. Conceivably we could have a `Link` class, containing further information such as probability of being clicked, or number of times clicked, but for now a pointer will do.
- Ultimately we want to have a class `Web` which contains a number of pages and their links. The web object will ultimately also contain information such as relative importance of the pages.

This application is a natural one for using pointers. When you click on a link on a web page you go from looking at one page in your browser to looking at another. You could implement this by having a pointer to a page, and clicking updates the value of this pointer.

Exercise 22.1. Make a class `Page` which initially just contains the name of the page. Write a method to display the page. Since we will be using pointers quite a bit, let this be the intended code for testing:

```
auto homepage = make_shared<Page>("My Home Page");
cout << "Homepage has no links yet:" << '\n';
cout << homepage->as_string() << '\n';
```

Next, add links to the page. A link is a pointer to another page, and since there can be any number of them, you will need a `vector` of them. Write a method `click` that follows the link. Intended code:

```
auto utexas = make_shared<Page>("University Home Page");
homepage->add_link(utexas);
auto searchpage = make_shared<Page>("google");
homepage->add_link(searchpage);
cout << homepage->as_string() << '\n';
```

Exercise 22.2. Add some more links to your homepage. Write a method `random_click` for the `Page` class. Intended code:

```
for (int iclick=0; iclick<20; ++iclick) {
```

```
    auto newpage = homepage->random_click();
    cout << "To: " << newpage->as_string() << '\n';
  }
```

How do you handle the case of a page without links?

22.2 Clicking around

Exercise 22.3. Now make a class `Web` which foremost contains a bunch (technically: a `vector`) of pages. Or rather: of pointers to pages. Since we don't want to build a whole internet by hand, let's have a method `create_random_links` which makes a random number of links to random pages. Intended code:

```
Web internet(netsize);
internet.create_random_links(avglinks);
```

Now we can start our simulation. Write a method `Web::random_walk` that takes a page, and the length of the walk, and simulates the result of randomly clicking that many times on the current page. (Current page. Not the starting page.)

Let's start working towards PageRank. First we see if there are pages that are more popular than others. You can do that by starting a random walk once on each page. Or maybe a couple of times.

Exercise 22.4. Apart from the size of your internet, what other design parameters are there for your tests? Can you give a back-of-the-envelope estimation of their effect?

Exercise 22.5. Your first simulation is to start on each page a number of times, and counts where that lands you. Intended code:

```
vector<int> landing_counts(internet.number_of_pages(),0);
for ( auto page : internet.all_pages() ) {
  for (int iwalk=0; iwalk<5; ++iwalk) {
    auto endpage = internet.random_walk(page,2*avglinks,tracing);
    landing_counts.at(endpage->global_ID())++;
  }
}
```

Display the results and analyze. You may find that you finish on certain pages too many times. What's happening? Fix that.

22.3 Graph algorithms

There are many algorithms that rely on gradually traversing the web. For instance, any graph can be *connected*. You test that by

- Take an arbitrary vertex v. Make a 'reachable set' $R \leftarrow \{v\}$.
- Now see where you can get from your reachable set:

$$\forall_{v\in V}\forall_{w \text{ neighbour of } v} : R \leftarrow R \cup \{w\}$$

- Repeat the previous step until R does not change anymore.

After this algorithm concludes, is R equal to your set of vertices? If so, your graph is called (fully) connected. If not, your graph has multiple *connected components*.

Exercise 22.6. Code the above algorithm, keeping track of how many steps it takes to reach each vertex w. This is the *Single Source Shortest Path* algorithm (for unweighted graphs).

The *diameter* is defined as the maximal shortest path. Code this.

22.4 Page ranking

The Pagerank algorithm now asks, if you keep clicking randomly, what is the distribution of how likely you are to wind up on a certain page. The way we calculate that is with a probability distribution: we assign a probability to each page so that the sum of all probabilities is one. We start with a random distribution:

Code:

```
ProbabilityDistribution
    random_state(internet.number_of_pages()
random_state.set_random();
cout << "Initial distribution: " <<
    random_state.as_string() << '\n';
```

Output
[google] pdfsetup:

```
Initial distribution:
    0:0.00, 1:0.02, 2:0.07,
    3:0.05, 4:0.06, 5:0.08,
    6:0.04, 7:0.04, 8:0.04,
    9:0.01, 10:0.07, 11:0.05,
    12:0.01, 13:0.04,
    14:0.08, 15:0.06,
    16:0.10, 17:0.06,
    18:0.11, 19:0.01,
```

Exercise 22.7. Implement a class `ProbabilityDistribution`, which stores a vector of floating point numbers. Write methods for:

- accessing a specific element,
- setting the whole distribution to random, and
- normalizing so that the sum of the probabilities is 1.
- a method rendering the distribution as string could be useful too.

Next we need a method that given a probability distribution, gives you the new distribution corresponding to performing a single click. (This is related to *Markov chains*; see HPC book [11], section 9.2.1.)

Exercise 22.8. Write the method

```
ProbabilityDistribution Web::globalclick
    ( ProbabilityDistribution currentstate );
```

Test it by

- start with a distribution that is nonzero in exactly one page;
- print the new distribution corresponding to one click;
- do this for several pages and inspect the result visually.

Then start with a random distribution and run a couple of iterations. How fast does the process converge? Compare the result to the random walk exercise above.

Exercise 22.9. In the random walk exercise you had to deal with the fact that some pages have no outgoing links. In that case you transitioned to a random page. That mechanism is lacking in the `globalclick` method. Figure out a way to incorporate this.

Let's simulate some simple 'search engine optimization' trick.

Exercise 22.10. Add a page that you will artificially made look important: add a number of pagesthat all link to this page, but no one links to them. (Because of the random clicking they will still sometimes be reached.)

Compute the rank of the artificially hyped page. Did you manage to trick Google into ranking this page high? How many links did you have to add?

Sample output:

```
Internet has 5000 pages
Top score: 109:0.0013, 3179:0.0012, 4655:0.0010, 3465:0.0009, 4298:0.0008
With fake pages:
Internet has 5051 pages
Top score: 109:0.0013, 3179:0.0012, 4655:0.0010, 5050:0.0010, 4298:0.0008
Hyped page scores at 4
```

22.5 Graphs and linear algebra

The probability distribution is essentially a vector. You can also represent the web as a matrix W with $w_{ij} = 1$ if page i links to page j. How can you interpret the `globalclick` method in these terms?

Exercise 22.11. Add the matrix representation of the `Web` object and reimplement the `globalclick` method. Test for correctness.

Do a timing comparison.

The iteration you did above to find a stable probability distribution corresponds to the 'power method' in linear algebra. Look up the Perron-Frobenius theory and see what it implies for page ranking.

Chapter 23

Redistricting

In this project you can explore 'gerrymandering', the strategic drawing of districts to give a minority population a majority of districts[1].

23.1 Basic concepts

We are dealing with the following concepts:

- A state is divided into census districts, which are given. Based on census data (income, ethnicity, median age) one can usually make a good guess as to the overall voting in such a district.
- There is a predetermined number of congressional districts, each of which consists of census districts. A congressional district is not a random collection: the census districts have to be contiguous.
- Every couple of years, to account for changing populations, the district boundaries are redrawn. This is known as redistricting.

There is considerable freedom in how redistricting is done: by shifting the boundaries of the (congressional) districts it is possible to give a population that is in the overall minority a majority of districts. This is known as *gerrymandering*.

For background reading, see `https://redistrictingonline.org/`.

To do a small-scale computer simulation of gerrymandering, we make some simplifying assumption.

- First of all, we dispense with census district: we assume that a district consists directly of voters, and that we know their affiliation. In practice one relies on proxy measures (such as income and education level) to predict affiliation.
- Next, we assume a one-dimensional state. This is enough to construct examples that bring out the essence of the problem:

  Consider a state of five voters, and we designate their votes as `AAABB`. Assigning them to three (contiguous) districts can be done as `AAA|B|B`, which has one 'A' district and two 'B' districts.
- We also allow districts to be any positive size, as long as the number of districts is fixed.

1. This project is obviously based on the Northern American political system. Hopefully the explanations here are clear enough. Please contact the author if you know of other countries that have a similar system.

23.2 Basic functions

23.2.1 Voters

We dispense with census districts, expressing everything in terms of voters, for which we assume a known voting behavior. Hence, we need a `Voter` class, which will record the voter ID and party affiliation. We assume two parties, and leave an option for being undecided.

Exercise 23.1. Implement a `Voter` class. You could for instance let ± 1 stand for `A/B`, and 0 for undecided.

```
cout << "Voter 5 is positive:" << '\n';
Voter nr5(5,+1);
cout << nr5.print() << '\n';

cout << "Voter 6 is negative:" << '\n';
Voter nr6(6,-1);
cout << nr6.print() << '\n';

cout << "Voter 7 is weird:" << '\n';
Voter nr7(7,3);
cout << nr7.print() << '\n';
```

23.2.2 Populations

Exercise 23.2. Implement a `District` class that models a group of voters.

- You probably want to create a district out of a single voter, or a vector of them. Having a constructor that accepts a string representation would be nice too.
- Write methods `majority` to give the exact majority or minority, and `lean` that evaluates whether the district overall counts as A party or B party.
- Write a `sub` method to creates subsets.

  ```
  District District::sub(int first,int last);
  ```

- For debugging and reporting it may be a good idea to have a method

  ```
  string District::print();
  ```

Code:

```
1     cout << "Making district with one B
      voter" << '\n';
2     Voter nr5(5,+1);
3     District nine( nr5 );
4     cout << ".. size: " << nine.size() <<
      '\n';
5     cout << ".. lean: " << nine.lean() <<
      '\n';
6     /* ... */
7     cout << "Making district ABA" << '\n';
8     District nine( vector<Voter>
9                    { {1,-1},{2,+1},{3,-1}
      } );
10    cout << ".. size: " << nine.size() <<
      '\n';
11    cout << ".. lean: " << nine.lean() <<
      '\n';
```

Output

```
[gerry] district:
Making district with one B voter
.. size: 1
.. lean: 1

Making district ABA
.. size: 3
.. lean: -1
```

Exercise 23.3. Implement a `Population` class that will initially model a whole state.

Code:

```
1 string pns( "-++--" );
2 Population some(pns);
3 cout << "Population from string " << pns
      << '\n';
4 cout << ".. size: " << some.size() <<
      '\n';
5 cout << ".. lean: " << some.lean() <<
      '\n';
6 Population group=some.sub(1,3);
7 cout << "sub population 1--3" << '\n';
8 cout << ".. size: " << group.size() <<
      '\n';
9 cout << ".. lean: " << group.lean() <<
      '\n';
```

Output

```
[gerry] population:
Population from string -++--
.. size: 5
.. lean: -1
sub population 1--3
.. size: 2
.. lean: 1
```

In addition to an explicit creation, also write a constructor that specifies how many people and what the majority is:

```
Population( int population_size,int majority,bool trace=false )
```

Use a random number generator to achieve precisely the indicated majority.

23.2.3 Districting

The next level of complication is to have a set of districts. Since we will be creating this incrementally, we need some methods for extending it.

Exercise 23.4. Write a class `Districting` that stores a `vector` of `District` objects. Write `size` and `lean` methods:

Code:

```
cout << "Making single voter population
    B" << '\n';
Population people( vector<Voter>{
    Voter(0,+1) } );
cout << ".. size: " << people.size() <<
    '\n';
cout << ".. lean: " << people.lean() <<
    '\n';

Districting gerry;
cout << "Start with empty districting:"
    << '\n';
cout << ".. number of districts: " <<
    gerry.size() << '\n';
```

Output

```
[gerry] gerryempty:
Making single voter population B
.. size: 1
.. lean: 1
Start with empty districting:
.. number of districts: 0
```

Exercise 23.5. Write methods to extend a `Districting`:

```
cout << "Add one B voter:" << '\n';
gerry = gerry.extend_with_new_district( people.at(0) );
cout << ".. number of districts: " << gerry.size() << '\n';
cout << ".. lean: " << gerry.lean() << '\n';
cout << "add A A:" << '\n';
gerry = gerry.extend_last_district( Voter(1,-1) );
gerry = gerry.extend_last_district( Voter(2,-1) );
cout << ".. number of districts: " << gerry.size() << '\n';
cout << ".. lean: " << gerry.lean() << '\n';

cout << "Add two B districts:" << '\n';
gerry = gerry.extend_with_new_district( Voter(3,+1) );
gerry = gerry.extend_with_new_district( Voter(4,+1) );
cout << ".. number of districts: " << gerry.size() << '\n';
cout << ".. lean: " << gerry.lean() << '\n';
```

23.3 Strategy

Now we need a method for districting a population:

```
Districting Population::minority_rules( int ndistricts );
```

Rather than generating all possible partitions of the population, we take an incremental approach (this is related to the solution strategy called *dynamic programming*):

- The basic question is to divide a population optimally over n districts;
- We do this recursively by first solving a division of a subpopulation over $n-1$ districts,
- and extending that with the remaining population as one district.

This means that you need to consider all the ways of having the 'remaining' population into one district, and that means that you will have a loop over all ways of splitting the population, outside of your recursion; see figure 23.1Multiple ways of splitting a populationfigure.23.1.

- For all $p = 0, \ldots n-1$ considering splitting the state into $0, \ldots, p-1$ and $p, \ldots, n-1$.
- Use the best districting of the first group, and make the last group into a single district.

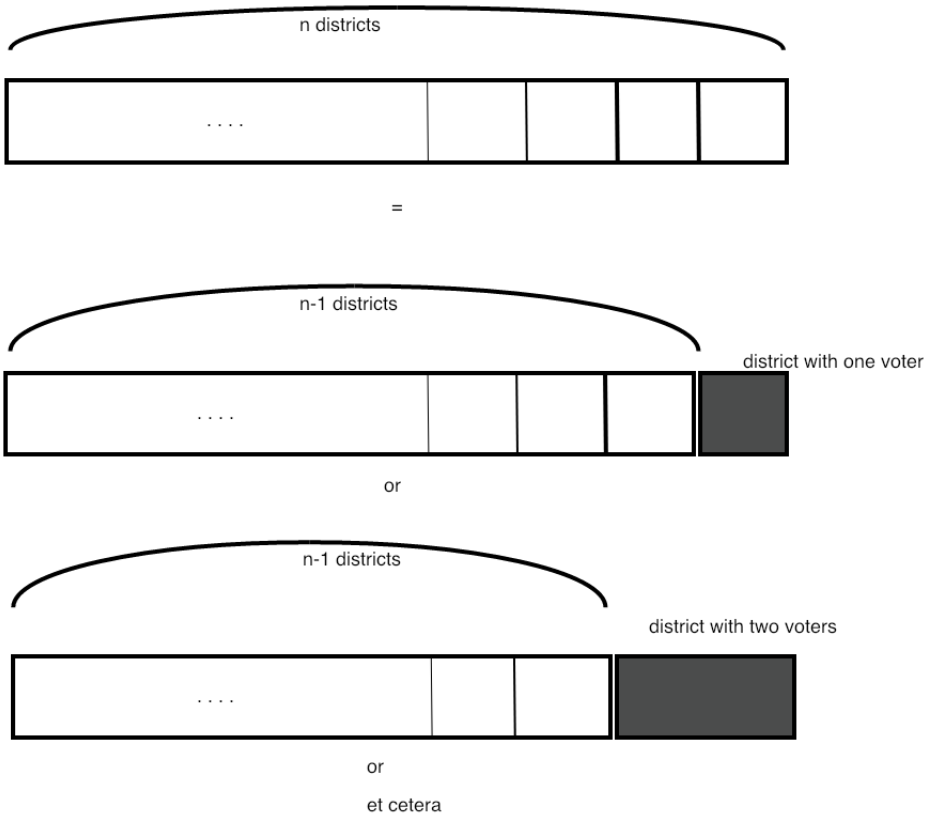


Figure 23.1: Multiple ways of splitting a population

- Keep the districting that gives the strongest minority rule, over all values of p.

You can now realize the above simple example:

```
AAABB => AAA|B|B
```

Exercise 23.6. Implement the above scheme.

Code:

```
1  Population five("+++--");
2  cout << "Redistricting population: " <<
       '\n'
3       << five.print() << '\n';
4  cout << ".. majority rule: "
5       << five.rule() << '\n';
6  int ndistricts{3};
7  auto gerry =
       five.minority_rules(ndistricts);
8  cout << gerry.print() << '\n';
9  cout << ".. minority rule: "
10      << gerry.rule() << '\n';
```

Output

```
[gerry] district5:

Redistricting population:
[0:+,1:+,2:+,3:-,4:-,]
.. majority rule: 1
[3[0:+,1:+,2:+,],[3:-,],[4:-,],]
.. minority rule: -1
```

Note: the range for p given above is not quite correct: for instance, the initial part of the population needs to be big enough to accommodate $n - 1$ voters.

Exercise 23.7. Test multiple population sizes; how much majority can you give party B while still giving party A a majority.

23.4 Efficiency: dynamic programming

If you think about the algorithm you just implemented, you may notice that the districtings of the initial parts get recomputed quite a bit. A strategy for optimizing for this is called *memoization*.

Exercise 23.8. Improve your implementation by storing and reusing results for the initial sub-populations.

In a way, we solved the program backward: we looked at making a district out of the last so-many voters, and then recursively solving a smaller problem for the first however-many voters. But in that process, we decided what is the best way to assign districts to the first 1 voter, first 2, first 3, et cetera. Actually, for more than one voter, say five voters, we found the result on the best attainable minority rule assigning these five voters to one, two, three, four districts.

The process of computing the 'best' districting forward, is known as *dynamic programming*. The fundamental assumption here is that you can use intermediate results and extend them, without having to reconsider the earlier problems.

Consider for instance that you've considered districting ten voters over up to five districts. Now the majority for eleven voters and five districts is the minimum of

- ten voters and five districts, and the new voter is added to the last district; or
- ten voters and four districts, and the new voter becomes a new district.

Exercise 23.9. Code a dynamic programming solution to the redistricting problem.

23.5 Extensions

The project so far has several simplifying assumptions.

- Congressional districts need to be approximately the same size. Can you put a limit on the ratio between sizes? Can the minority still gain a majority?

Exercise 23.10. The biggest assumption is of course that we considered a one-dimensional state. With two dimensions you have more degrees of freedom of shaping the districts. Implement a two-dimensional scheme; use a completely square state, where the census districts form a regular grid. Limit the shape of the congressional districts to be convex.

The *efficiency gap* is a measure of how 'fair' a districting of a state is.

Exercise 23.11. Look up the definition of efficiency gap (and 'wasted votes'), and implement it in your code.

23.6 Ethics

The activity of redistricting was intended to give people a fair representation. In its degenerate form of Gerrymandering this concept of fairness is violated because the explicit goal is to give the minority a majority of votes. Explore ways that this unfairness can be undone.

In your explorations above, the only characteristic of a voter was their preference for party A or B. However, in practice voters can be considered part of communities. The Voting Rights Act is concerned about 'minority vote dilution'. Can you show examples that a color-blind districting would affect some communities negatively?

Chapter 24

Amazon delivery truck scheduling

This section contains a sequence of exercises that builds up to a simulation of delivery truck scheduling.

24.1 Problem statement

Scheduling the route of a delivery truck is a well-studied problem. For instance, minimizing the total distance that the truck has to travel corresponds to the *Traveling Salesman Problem (TSP)*—textbf. However, in the case of *Amazon delivery truck* scheduling the problem has some new aspects:

- A customer is promised a window of days when delivery can take place. Thus, the truck can split the list of places into sublists, with a shorter total distance than going through the list in one sweep.
- Except that *Amazon prime* customers need their deliveries guaranteed the next day.

24.2 Coding up the basics

Before we try finding the best route, let's put the basics in place to have any sort of route at all.

24.2.1 Address list

You probably need a class `Address` that describes the location of a house where a delivery has to be made.

- For simplicity, let give a house (i, j) coordinates.
- We probably need a `distance` function between two addresses. We can either assume that we can travel in a straight line between two houses, or that the city is build on a grid, and you can apply the so-called *Manhattan distance*.
- The address may also require a field recording the last possible delivery date.

Exercise 24.1. Code a class `Address` with the above functionality, and test it.

Code:

```
1     Address one(1.,1.),
2       two(2.,2.);
3     cerr << "Distance: "
4          << one.distance(two)
5          << '\n';
```

Output

```
[amazon] address:

Address
Distance: 1.41421
.. address
Address 1 should be closest to
    the depot. Check: 1

Route from depot to depot:
    (0,0) (2,0) (1,0) (3,0)
    (0,0)
 has length 8: 8
Greedy scheduling:  (0,0) (1,0)
    (2,0) (3,0) (0,0)
 should have length 6: 6

Square5
Travel in order: 24.1421
Square route:  (0,0) (0,5)
    (5,5) (5,0) (0,0)
 has length 20
.. square5

Original list:  (0,0) (-2,0)
    (-1,0) (1,0) (2,0) (0,0)
 length=8
flip middle two addresses:
    (0,0) (-2,0) (1,0) (-1,0)
    (2,0) (0,0)
 length=12
better:  (0,0) (1,0) (-2,0)
    (-1,0) (2,0) (0,0)
 length=10

Hundred houses
Route in order has length
    25852.6
TSP based on mere listing has
    length: 2751.99 over naive
    25852.6
Single route has length: 2078.43
 .. new route accepted with
    length 2076.65
Final route has length 2076.65
    over initial 2078.43
TSP route has length 1899.4
    over initial 2078.43

Two routes
Route1:  (0,0) (2,0) (3,2)
    (2,3) (0,2) (0,0)
route2:  (0,0) (3,1) (2,1)
    (1,2) (1,3) (0,0)
total length 19.6251
start with 9.88635,9.73877
Pass 0
.. down to 9.81256,8.57649
Pass 1
Pass 2
Pass 3
Pass 4
TSP Route1:  (0,0) (3,1) (3,2)
    (2,3) (0,2) (0,0)
route2:  (0,0) (2,0) (2,1)
    (1,2) (1,3) (0,0)
total length 18.389
```

Next we need a class `AddressList` that contains a list of addresses.

Exercise 24.2. Implement a class `AddressList`; it probably needs the following methods:

- `add_address` for constructing the list;
- `length` to give the distance one has to travel to visit all addresses in order;
- `index_closest_to` that gives you the address on the list closest to another address, presumably not on the list.

24.2.2 Add a depot

Next, we model the fact that the route needs to start and end at the depot, which we put arbitrarily at coordinates $(0, 0)$. We could construct an `AddressList` that has the depot as first and last element, but that may run into problems:

- If we reorder the list to minimize the driving distance, the first and last elements may not stay in place.
- We may want elements of a list to be unique: having an address twice means two deliveries at the same address, so the `add_address` method would check that an address is not already in the list.

We can solve this by making a class `Route`, which inherits from `AddressList`, but the methods of which leave the first and last element of the list in place.

24.2.3 Greedy construction of a route

Next we need to construct a route. Rather than solving the full TSP, we start by employing a *greedy search* strategy:

> Given a point, find the next point by some local optimality test, such as shortest distance. Never look back to revisit the route you have constructed so far.

Such a strategy is likely to give an improvement, but most likely will not give the optimal route.

Let's write a method

```
Route::Route greedy_route();
```

that constructs a new address list, containing the same addresses, but arranged to give a shorter length to travel.

Exercise 24.3. Write the `greedy_route` method for the `AddressList` class.

1. Assume that the route starts at the depot, which is located at $(0, 0)$. Then incrementally construct a new list by:
2. Maintain an `Address` variable `we_are_here` of the current location;
3. repeatedly find the address closest to `we_are_here`.

Extend this to a method for the `Route` class by working on the subvector that does not contain the final element.

Test it on this example:

Code:

```
1     Route deliveries;
2     deliveries.add_address( Address(0,5)
      );
3     deliveries.add_address( Address(5,0)
      );
4     deliveries.add_address( Address(5,5)
      );
5     cerr << "Travel in order: " <<
      deliveries.length() << '\n';
6     assert( deliveries.size()==5 );
7     auto route =
      deliveries.greedy_route();
8     assert( route.size()==5 );
9     auto len = route.length();
10    cerr << "Square route: " <<
      route.as_string()
11          << "\n has length " << len <<
      '\n';
```

Output

```
[amazon] square5:

Travel in order: 24.1421
Square route:  (0,0) (0,5)
    (5,5) (5,0) (0,0)
 has length 20
```

Reorganizing a list can be done in a number of ways.

- First of all, you can try to make the changes in place. This runs into the objection that maybe you want to save the original list; also, while swapping two elements can be done with the `insert` and `erase` methods, more complicated operations are tricky.
- Alternatively, you can incrementally construct a new list. Now the main problem is to keep track of which elements of the original have been processed. You could do this by giving each address a boolean field `done`, but you could also make a copy of the input list, and remove the elements that have been processed. For this, study the `erase` method for `vector` objects.

24.3 Optimizing the route

The above suggestion of each time finding the closest address is known as a *greedy search* strategy. It does not give you the optimal solution of the TSP. Finding the optimal solution of the TSP is hard to program – you could do it recursively – and takes a lot of time as the number of addresses grows. In fact, the TSP is probably the most famous of the class of *NP-hard* problems, which are generally believed to have a running time that grows faster than polynomial in the problem size.

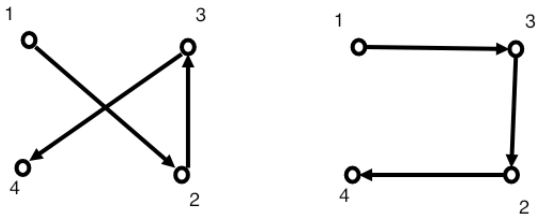


Figure 24.1: Illustration of the 'opt2' idea of reversing part of a path

However, you can approximate the solution heuristically. One method, the Kernighan-Lin algorithm [20], is based on the *opt2* idea: if you have a path that 'crosses itself', you can make it shorter by reversing

part of it. Figure 24.1Illustration of the 'opt2' idea of reversing part of a pathfigure.24.1 shows that the path $1 - -2 - -3 - -4$ can be made shorter by reversing part of it, giving $1 - -3 - -2 - -4$. Since recognizing where a path crosses itself can be hard, or even impossible for graphs that don't have Cartesian coordinates associated, we adopt a scheme where we try all possible reversals:

```
for all nodes m<n on the path [1..N]:
  make a new route from
    [1..m-1] + [m--n].reversed + [n+1..N]
  if the new route is shorter, keep it
```

Exercise 24.4. Code the opt2 heuristic: write a method to reverse part of the route, and write the loop that tries this with multiple starting and ending points. Try it out on some simple test cases to convince you that your code works as intended.

Let's explore issues of complexity. (For an introduction to complexity calculations, see HPC book [11], section 19.) The TSP is one of a class of *NP complete* problems, which very informally means that there is no better solution than trying out all possibilities.

Exercise 24.5. What is the runtime complexity of the heuristic solution using opt2? What would the runtime complexity be of finding the best solution by considering all possibilities? Make a very rough estimation of runtimes of the two strategies on some problem sizes: $N = 10, 100, 1000, \ldots$.

Exercise 24.6. Earlier you had programmed the greedy heuristic. Compare the improvement you get from the opt2 heuristic, starting both with the given list of addresses, and with a greedy traversal of it.

For realism, how many addresses do you put on your route? How many addresses would a delivery driver do on a typical day?

24.4 Multiple trucks

If we introduce multiple delivery trucks, we get the 'Multiple Traveling Salesman Problem' [6]. With this we can module both the cases of multiple trucks being out on delivery on the same day, or one truck spreading deliveries over multiple days. For now we don't distinguish between the two.

The first question is how to divide up the addresses.

1. We could split the list in two, using some geometric test. This is a good model for the case where multiple trucks are out on the same day. However, if we use this as a model for the same truck being out on multiple days, we are missing the fact that new addresses can be added on the first day, messing up the neatly separated routes.
2. Thus it may in fact be reasonable to assume that all trucks get an essentially random list of addresses.

Can we extend the opt2 heuristic to the case of multiple paths? For inspiration take a look at figure 24.2Extending the 'opt2' idea to multiple pathsfigure.24.2: instead of modifying one path, we could switch bits out bits between one path and another. When you write the code, take into account that the other path may be running backwards! This means that based on split points in the first and second path you know have four resulting modified paths to consider.

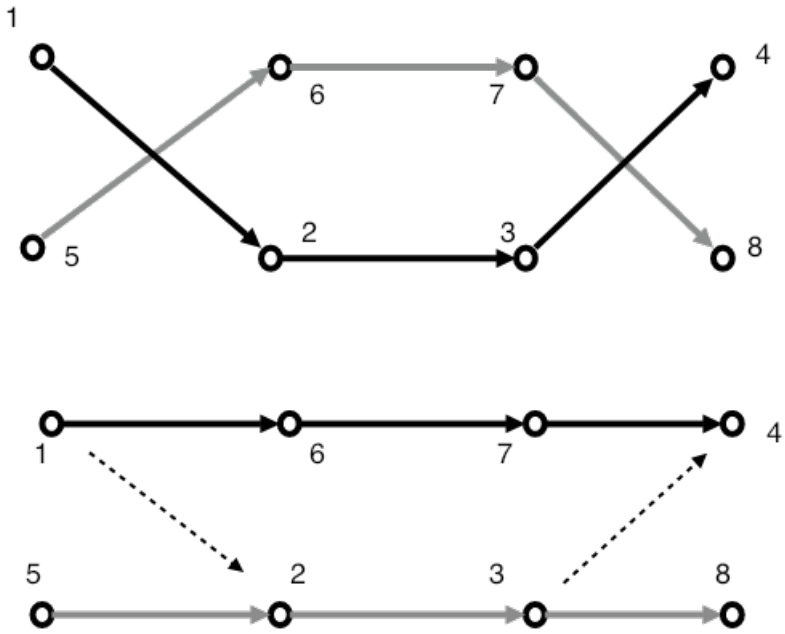


Figure 24.2: Extending the ‘opt2’ idea to multiple paths

Exercise 24.7. Write a function that optimizes two paths simultaneously using the multi-path version of the opt2 heuristic. For a test case, see figure 24.3Multiple paths test casefigure.24.3.

You have quite a bit of freedom here:

- The start points of the two segments should be chosen independently;
- the lengths can be chosen independently, but need not; and finally
- each segment can be reversed.

More flexibility also means a longer runtime of your program. Does it pay off? Do some tests and report results.

Based on the above description there will be a lot of code duplication. Make sure to introduce functions and methods for various operations.

24.5 Amazon prime

In section 24.4Multiple truckssection.24.4 you made the assumption that it doesn’t matter on what day a package is delivered. This changes with *Amazon prime*, where a package has to be delivered guaranteed on the next day.

Exercise 24.8. Explore a scenario where there are two trucks, and each have a number of addresses that can not be exchanged with the other route. How much longer is the total distance? Experiment with the ratio of prime to non-prime addresses.

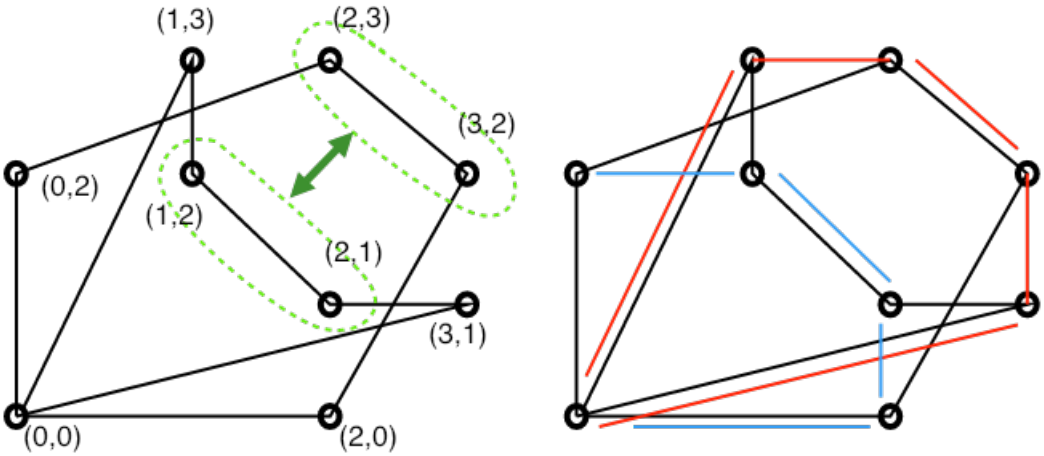


Figure 24.3: Multiple paths test case

24.6 Dynamicism

So far we have assumed that the list of addresses to be delivered to is given. This is of course not true: new deliveries will need to be scheduled continuously.

Exercise 24.9. Implement a scenario where every day a random number of new deliveries is added to the list. Explore strategies and design choices.

24.7 Ethics

People sometimes criticize Amazon's labor policies, including regarding its drivers. Can you make any observations from your simulations in this respect?

Chapter 25

High performance linear algebra

Linear algebra is fundamental to much of computational science. Applications involving Partial Diffential Equations (PDEs) come down to solving large systems of linear equation; solid state physics involves large eigenvalue systems. But even outside of engineering applications linear algebra is important: the major computational part of Deep Learning (DL) networks involves matrix-matrix multiplications.

Linear algebra operations such as the matrix-matrix product are easy to code in a naive way. However, this does not lead to high performance. In these exercises you will explore the basics of a strategy for high performance.

25.1 Mathematical preliminaries

The matrix-matrix product $C \leftarrow A \cdot B$ is defined as

$$\forall_{ij} : c_{ij} \leftarrow \sum_k a_{ik} b_{kj}.$$

Straightforward code for this would be:

```
for (i=0; i<a.m; i++)
  for (j=0; j<b.n; j++)
    s = 0;
    for (k=0; k<a.n; k++)
      s += a[i,k] * b[k,j];
    c[i,j] = s;
```

However, this is not the only way to code this operation. The loops can be permuted, giving a total of six implementations.

Exercise 25.1. Code one of the permuted algorithms and test its correctness. If the reference algorithm above can be said to be 'inner-product based', how would you describe your variant?

Yet another implementation is based on a block partitioning. Let A, B, C be split on 2×2 block form:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \qquad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \qquad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

Then

$$\begin{aligned} C_{11}&=A_{11}B_{11}+A_{12}B_{21},\\ C_{12}&=A_{11}B_{12}+A_{12}B_{22},\\ C_{21}&=A_{21}B_{11}+A_{22}B_{21},\\ C_{22}&=A_{21}B_{12}+A_{22}B_{22} \end{aligned} \tag{25.1}$$

Convince yourself that this actually computes the same product $C = A \cdot B$. For more on block algorithms, see HPC book [11], section 5.3.6.

Exercise 25.2. Write a matrix class with a multiplication routine:

```
Matrix Matrix::MatMult( Matrix other );
```

First implement a traditional matrix-matrix multiplication, then make it recursive. For the recursive algorithm you need to implement sub-matrix handling: you need to extract submatrices, and write a submatrix back into the surrounding matrix.

25.2 Matrix storage

The simplest way to store an $M \times N$ matrix is as an array of length MN. Inside this array we can decide to store the rows end-to-end, or the columns. While this decision is obviously of practical importance for a library, from a point of performance it makes no difference.

Remark 17 *Historically, linear algebra software such as the Basic Linear Algebra Subprograms (BLAS) has used columnwise storage, meaning that the location of an element (i, j) is computed as $i + j \cdot M$ (we will use zero-based indexing throughout this project, both for code and mathematical expressions.) The reason for this stems from the origins of the BLAS in the Fortran language, which uses column-major ordering of array elements. On the other hand, static arrays (such as `x[5][6][7]`) in the C/C++ languages have row-major ordering, where element (i, j) is stored in location $j + i \cdot N$.*

Above, you saw the idea of block algorithms, which requires taking submatrices. For efficiency, we don't want to copy elements into a new array, so we want the submatrix to correspond to a subarray.

Now we have a problem: only a submatrix that consists of a sequence of columns is contiguous. The formula $i + j \cdot M$ for location of element (i, j) is no long correct if the matrix is a subblock of a larger matrix.

For this reason, linear algebra software describes a submatrix by three parameters M, N, LDA, where 'LDA' stands for 'leading dimension of A' (see BLAS [17], and Lapack [2]). This is illustrated in figure 25.1Submatrix out of a matrix, with M, N, LDA of the submatrix indicatedfigure.25.1.

Exercise 25.3. In terms of M, N, LDA, what is the location of the (i, j) element?

Implementationwise we also have a problem. If we use `std::vector` for storage, it is not possible to take subarrays, since C++ insists that a vector has its own storage. The solution is to use `span`; section **??**.

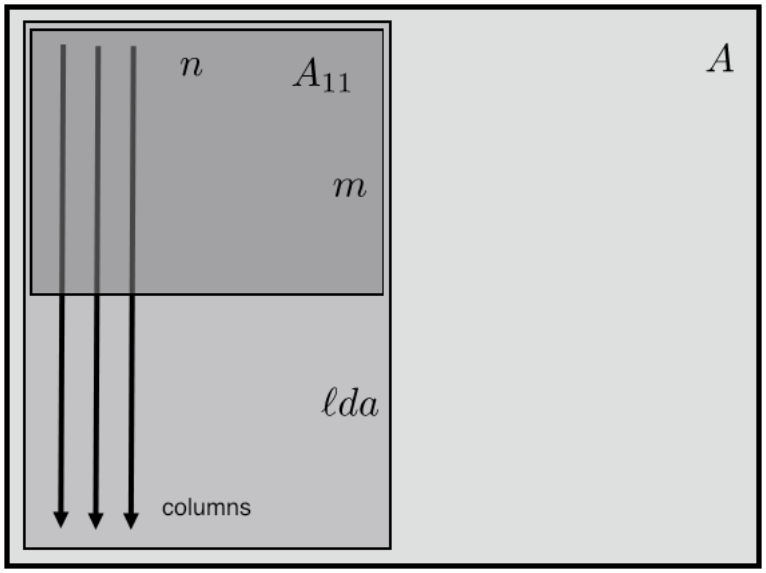


Figure 25.1: Submatrix out of a matrix, with M, N, LDA of the submatrix indicated

We could have two types of matrices: top level matrices that store a `vector<double>`, and submatrices that store a `span<double>`, but that is a lot of complication. It could be done using `std::variant` (section **??**), but let's not.

Instead, let's adopt the following idiom, where we create a vector at the top level, and then create matrices from its memory.

```
// example values for M,LDA,N
M = 2; LDA = M+2; N = 3;
// create a vector to contain the data
vector<double> one_data(LDA*N,1.);
// create a matrix using the vector data
Matrix one(M,LDA,N,one_data.data());
```

(If you have not previously programmed in C, you need to get used to the `double*` mechanism. See section **??**.)

Exercise 25.4. Start implementing the `Matrix` class with a constructor

```
Matrix::Matrix(int m,int lda,int n,double *data)
```

and private data members:

```
private:
  int m,n,lda;
  span<double> data;
```

Write a method

```
double& Matrix::at(int i,int j);
```

that you can use as a safe way of accessing elements.

Let's start with simple operations.

Exercise 25.5. Write a method for adding matrices. Test it on matrices that have the same M, N, but different LDA.

Use of the `at` method is great for debugging, but it is not efficient. Use the preprocessor (chapter **??**) to introduce alternatives:

```
#ifdef DEBUG
  c.at(i,j) += a.at(i,k) * b.at(k,j)
#else
  cdata[ /* expression with i,j */ ] += adata[ ... ] * bdata[ ... ]
#endif
```

where you access the data directly with

```
auto get_double_data() {
  double *adata;
  adata = data.data();
  return adata;
};
```

Exercise 25.6. Implement this. Use a cpp `#define` macro for the optimized indexing expression. (See section **??**.)

25.2.1 Submatrices

Next we need to support constructing actual submatrices. Since we will mostly aim for decomposition in 2×2 block form, it is enough to write four methods:

```
Matrix Left(int j);
Matrix Right(int j);
Matrix Top(int i);
Matrix Bot(int i);
```

where, for instance, `Left(5)` gives the columns with $j < 5$.

Exercise 25.7. Implement these methods and test them.

25.3 Multiplication

You can now write a first multiplication routine, for instance with a prototype

```
void Matrix::MatMult( Matrix& other,Matrix& out );
```

Alternatively, you could write

```
Matrix Matrix::MatMult( Matrix& other );
```

but we want to keep the amount of creation/destruction of objects to a minimum.

25.3.1 One level of blocking

Next, write

```
void Matrix::BlockedMatMult( Matrix& other,Matrix& out );
```

which uses the 2×2 form above.

25.3.2 Recursive blocking

The final step is to make the blocking recursive.

Exercise 25.8. Write a method

```
void RecursiveMatMult( Matrix& other,Matrix& out );
```

which

- Executes the 2×2 block product, using again `RecursiveMatMult` for the blocks.
- When the block is small enough, use the regular `MatMult` product.

25.4 Performance issues

If you experiment a little with the cutoff between the regular and recursive matrix-matrix product, you see that you can get good factor of performance improvement. Why is this?

The matrix-matrix product is a basic operation in scientific computations, and much effort has been put into optimizing it. One interesting fact is that it is just about the most optimizable operation under the sum. The reason for this is, in a nutshell, that it involves $O(N^3)$ operations on $O(N^2)$ data. This means that, in principle each element fetched will be used multiple times, thereby overcoming the *memory bottleneck*.

To understand performance issues relating to hardware, you need to do some reading. Section HPC book [11], section 1.3.4 explains the crucial concept of a *cache*.

Exercise 25.9. Argue that the naive matrix-matrix product implementation is unlikely actually to reuse data.

Explain why the recursive strategy does lead to data reuse.

Above, you set a cutoff point for when to switch from the recursive to the regular product.

Exercise 25.10. Argue that continuing to recurse will not have much benefit once the product is contained in the cache. What are the cache sizes of your processor?

Do experiments with various cutoff points. Can you relate this to the cache sizes?

25.4.1 Parallelism (optional)

The four clauses of equation 25.1Mathematical preliminariesequation.25.1.1 target independent areas in the C matrix, so they could be executed in parallel on any processor that has at least four cores.

Explore the OpenMP library to parallelize the `BlockedMatMult`.

25.4.2 Comparison (optional)

The final question is: how close are you getting to the best possible speed? Unfortunately you are still a way off. You can explore that as follows.

Your computer is likely to have an optimized implementation, accessible through:

```
#include <cblas.h>

cblas_dgemm
  ( CblasColMajor, CblasNoTrans, CblasNoTrans,
    m,other.n,n, alpha,adata,lda,
    bdata,other.lda,
    beta,cdata,out.lda);
```

which computes $C \leftarrow \alpha A \cdot B + \beta C$.

Exercise 25.11. Use another cpp conditional to implement `MatMult` through a call to `cblas_dgemm`. What performance do you now get?

You see that your recursive implementation is faster than the naive one, but not nearly as fast as the CBlas one. This is because

- the CBlas implementation is probably based on an entirely different strategy [14], and
- it probably involves a certain amount of assembly coding.

Chapter 26

The Great Garbage Patch

This section contains a sequence of exercises that builds up to a *cellular automaton* simulation of turtles in the ocean, garbage that is deadly to them, and ships that clean it up. To read more about this: https://theoceancleanup.com/.

Thanks to Ernesto Lima of TACC for the idea and initial code of this exercise.

26.1 Problem and model solution

There is lots of plastic floating around in the ocean, and that is harmful to fish and turtles and cetaceans. Here you get to model the interaction between

- Plastic, randomly located;
- Turtles that swim about; they breed slowly, and they die from ingesting plastic;
- Ships that sweep the ocean to remove plastic debris.

The simulation method we use is that of a *cellular automaton*:

- We have a grid of cells;
- Each cell has a 'state' associated with it, namely, it can contain a ship, a turtle, or plastic, or be empty; and
- On each next time step, the state of a cell is a simple function of the states of that cell and its immediate neighbors.

The purpose of this exercise is to simulate a number of time steps, and explore the iteraction between parameters: with how much gargage will turtles die out, how many ships are enough to protect the turtles.

26.2 Program design

The basic idea is to have an `ocean` object that is populated with turtles, trash, and ships. Your simulation will let the ocean undergo a number of time steps:

```
for (int t=0; t<time_steps; t++)
  ocean.update();
```

Ultimately your purpose is to investigate the development of the turtle population: is it stable, does it die out?

While you can make a 'hackish' solution to this problem, partly you will be judged on your use of modern/clean C++ programming techniques. A number of suggestions are made below.

26.2.1 Grid update

Here is a point to be aware of. Can you see what's wrong with with doing an update entirely in-place:

```
for ( i )
  for ( j )
    cell(i,j) = f( cell(i,j), .... other cells ... );
```

?

26.3 Testing

It can be complicated to test this program for correctness. The best you can do is to try out a number of scenarios. For that it's best to make your program input flexible: use the `cxxopts` package [8], and drive your program from a shell script.

Here is a list of things you can test.

1. Start with only a number of ships; check that after 1000 time steps you still have the same number.
2. Likewise with turtles; if they don't breed and don't die, check that their number stays constant.
3. With only ships and trash, does it all get swept?
4. With only turtles and trash, do they all die off?

It is harder to test that your turtles and ships don't 'teleport' around, but only move to contiguous cells. For that, use visual inspection; see section 26.3.1.

26.3.1 Animated graphics

The output of this program is a prime candidate for visualization. In fact, some test ('make sure that turtles don't teleport') are hard to do other than by looking at the output. Start by making an ascii rendering of the ocean grid, as in figure 26.1.

It would be better to have some sort of animated output. However, not all programming languages generate visual output equally easily. There are very powerful video/graphics libraries in C++, but these are also hard to use. There is a simpler way out.

For simple output such as this program yields, you can make a simple low-budget animation. Every *terminal emulator* under the sun supports *VT100 cursor control* [1]: you can send certain magic output to your screen to control cursor positioning.

In each time step you would

1. Send the cursor to the home position, by this magic output:

```
#include <cstdio>
    /* ... */
    // ESC [ i ; j H
    printf( "%c[0;0H",(char)27);
```

2. Display your grid as in figure 26.1;
3. Sleep for a fraction of a second; see section **??**.

1. https://vt100.net/docs/vt100-ug/chapter3.html

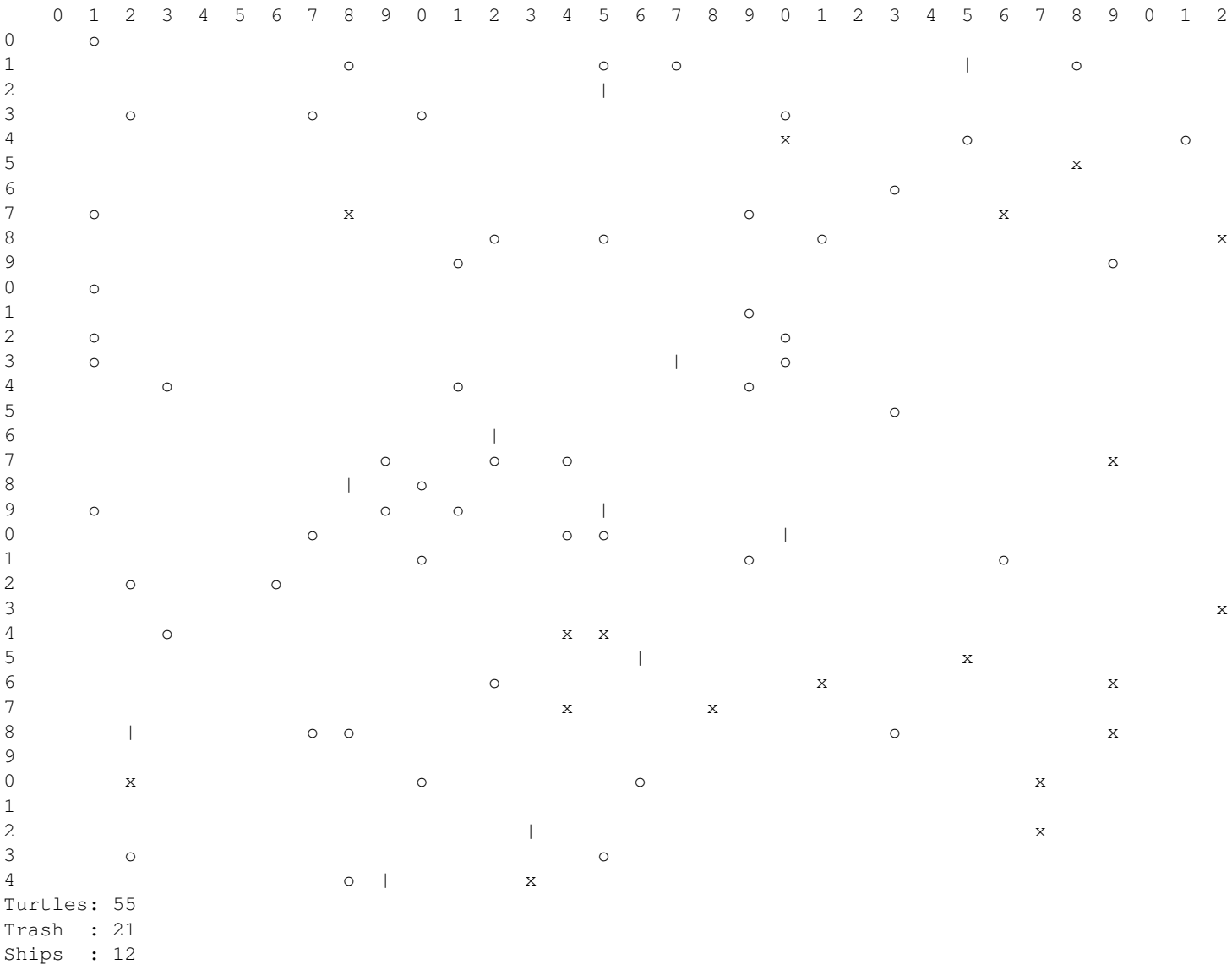


Figure 26.1: Ascii art printout of a time step

26.4 Modern programming techniques

26.4.1 Object oriented programming

While there is only have one ocean, you should still make an `ocean` class, rather than having a global array object. All functions are then methods of the single object you create of that class.

26.4.2 Data structure

We lose very little generality by ignoring the depth of the ocean and the shape of coastlines, and model the ocean as a 2D grid.

If you write an indexing function `cell(i,j)` you can make your code largely independent of the actual data structure chosen. Argue why `vector<vector<int>>` is not the best storage.

1. What do you use instead?
2. When you have a working code, can you show by timing that your choice is indeed superior?

26.4.3 Cell types

Having 'magic numbers' in your code (0 =empty, 1 =turtle, et cetera) is not elegant. Make a **`enum`** or **`enum class`** (see section 2.4Enums and enum classessection.2.4) so that you have names for what's in your cells:

```
cell(i,j) = occupy::turtle;
```

If you want to print out your ocean, it might be nice if you can directly **`cout`** your cells:

```
for (int i=0; i<iSize; ++i) {
  cout << i%10 << " ";
  for (int j=0; j<jSize; ++j) {
    cout << setw(hs) << cell(i,j);
  }
  cout << '\n';
}
```

26.4.4 Ranging over the ocean

It is easy enough to write a loop as

```
for (int i=0; i<iSize; i++)
  for (int j=0; j<jSize; j++)
    ... cell(i,j) ...
```

However, it may not be a good idea to always sweep over your domain so orderly. Can you implement this:

```
for ( auto [i,j] : permuted_indices() ) {
    ... cell(i,j) ...
```

? See section **??** about *structured binding*.

Likewise, if you need to count how many pieces of trash there are around a turtle, can you get this code to work:

```
int count_around( int ic,int jc,occupy typ ) const {
  int count=0;
  for ( auto [i,j] : neighbors(ic,jc) ) {
    if (cell(i,j) == typ)
      ++count;
  }
  return count;
};
```

26.4.5 Random numbers

For the random movement of ships and turtles you need a random number generator. Do not use the old C generator, but the new `random` one; section **??**.

Try to find a solution so that you use exactly one generator for all places where you need random numbers. Hint: make the generator `static` in your class.

26.5 Explorations

Instead of having the ships move randomly, can you give them a preferential direction to the closest garbage patch? Does this improve the health of the turtle population?

Can you account for the relative size of ships and turtles by having a ship occupy a 2×2 block in your grid?

So far you have let trash stay in place. What if there are ocean currents? Can you make the trash 'sticky' so that trash particles start moving as a patch if they touch?

Turtles eat sardines. (No they don't.) What happens to the sardine population if turtles die out? Can you come up with parameter values that correspond to a stable ecology or a de-stabilized one?

26.5.1 Code efficiency

Investigate whether your implementation of the `enum` in section 26.4.3 has any effect on timing. Parse the fine print of section 2.4Enums and enum classessection.2.4.

You may remark that ranging over a largely empty ocean can be pretty inefficient. You could contemplate keeping an 'active list' of where the turtles et cetera are located, and only looping over that. How would you implement that? Do you expect to see a difference in timing? Do you actually?

How is runtime affected by choosing a vector-of-vectors implementation for the ocean; see section 26.4.2.

Chapter 27

Graph algorithms

In this project you will explore some common graph algorithms, and their various possible implementations. The main theme here will be that the common textbook exposition of algorithms is not necessarily the best way to phrase them computationally.

As background knowledge for this project, you are encouraged to read HPC book [11], chapter 9; for an elementary tutorial on graphs, see HPC book [11], chapter 24.

27.1 Traditional algorithms

We first implement the 'textbook' formulations of two *Single Source Shortest Path (SSSP)* algorithms: on unweighted and then on weighted graphs. In the next section we will then consider formulations that are in terms of linear algebra.

In order to develop the implementations, we start with some necessary preliminaries,

27.1.1 Code preliminaries

27.1.1.1 Adjacency graph

We need a class `Dag` for a Directed Acyclic Graph (DAG):

```
class Dag {
private:
  vector< vector<int> > dag;
public:
  // Make Dag of 'n' nodes, no edges for now.
  Dag( int n )
    : dag( vector< vector<int> >(n) ) {};
```

It's probably a good idea to have a function

```
const auto& neighbors( int i ) const { return dag.at(i); };
```

that, given a node, returns a list of the neighbors of that node.

Exercise 27.1. Finish the `Dag` class. In particular, add a method to generate example graphs:

- For testing the 'circular' graph is often useful: connect edges

$$0 \to 1 \to \cdots \to N-1 \to 0.$$

- It may also be a good idea to have a graph with random edges.

Write a method that displays the graph.

27.1.1.2 Node sets

The classic formulation of SSSP algorithms, such as the *Dijkstra shortest path algorithm* (see HPC book [11], section 9.1.3) uses sets of nodes that are gradually built up or depleted.

You could implement that as a vector:

```
vector< int > set_of_nodes(nnodes);
for ( int inode=0; inode<nnodes; inode++)
  // mark inode as distance unknown:
  set_of_nodes.at(inode) = inf;
```

where you use some convention, such as negative distance, to indicate that a node has been removed from the set.

However, C++ has an actual `set` container with methods for adding an element, finding it, and removing it; see section **??**. This makes for a more direct expression of our algorithms. In our case, we'd need a set of int/int or int/float pairs, depending on the graph algorithm. (It is also possible to use a `map`, using an int as lookup key, and int or float as values.)

For the unweighted graph we only need a set of finished nodes, and we insert node 0 as our starting point:

```
using node_info = std::pair<unsigned,unsigned>;
std::set< node_info > distances;
distances.insert( {0,0} );
```

For Dijkstra's algorithm we need both a set of finished nodes, and nodes that we are still working on. We again set the starting node, and we set the distance for all unprocessed nodes to infinity:

```
const unsigned inf = std::numeric_limits<unsigned>::max();
using node_info = std::pair<unsigned,unsigned>;
std::set< node_info > distances,to_be_done;

to_be_done.insert( {0,0} );
for (unsigned n=1; n<graph_size; ++n)
  to_be_done.insert( {n,inf} );
```

(Why do we need that second set here, while it was not necessary for the unweighted graph case?)

Exercise 27.2. Write a code fragment that tests if a node is in the `distances` set.

- You can of course write a loop for this. In that case know that iterating over a set gives you the key/value pairs. Use *structured bindings*; section **??**.
- But it's better to use an 'algorithm', in the technical sense of 'algorithms built into the standard

library'. In this case, `find`.
- ... except that with `find` you have to search for an exact key/value pair, and here you want to search: 'is this node in the `distances` set with whatever value'. Use the `find_if` algorithm; section **??**.

27.1.2 Level set algorithm

We start with a simple algorithm: the SSSP algorithm in an unweighted graph; see HPC book [11], section 9.1.1 for details. Equivalently, we find level sets in the graph.

For unweighted graphs, the distance algorithm is fairly simple. Inductively:

- Assume that we have a set of nodes reachable in at most n steps,
- then their neighbors (that are not yet in this set) can be reached in $n+1$ steps.

The algorithm outline is

```
for (;;) {
  if (distances.size()==graph_size) break;
  /*
   * Loop over all nodes that are already done
   */
  for ( auto [node,level] : distances ) {
    /*
     * set neighbors of the node to have distance 'level + 1'
     */
    const auto& nbors = graph.neighbors(node);
    for ( auto n : nbors ) {
      /*
       * See if 'n' has a known distance,
       * if not, add to 'distances' with level+1
       */
/* ... */
        {
          cout << "node " << n << " level " << level+1 << '\n';
          distances.insert( {n,level+1} );
        }
```

Exercise 27.3. Finish the program that computes the SSSP algorithm and test it.

This code has an obvious inefficiency: for each level we iterate through all finished nodes, even if all their neighbors may already have been processed.

Exercise 27.4. Maintain a set of 'current level' nodes, and only investigate these to find the next level. Time the two variants on some large graphs.

27.1.3 Dijkstra's algorithm

In Dijkstra's algorithm we maintain both a set of nodes for which the shortest distance has been found, and one for which we are still determining the shortest distance. Note: a tentative shortest distance for a node may be updated several times, since there may be multiple paths to that node. The 'shortest' path in terms of weights may not be the shortest in number of edges traversed!

The main loop now looks something like:

```
for (;;) {
  if (to_be_done.size()==0) break;
  /*
   * Find the node with least distance
   */
  /* ... */
  cout << "min: " << nclose << " @ " << dclose << '\n';
  /*
   * Move that node to done,
   */
  to_be_done.erase(closest_node);
  distances.insert( *closest_node );
  /*
   * set neighbors of nclose to have that distance + 1
   */
  const auto& nbors = graph.neighbors(nclose);
  for ( auto n : nbors ) {
    // find 'n' in distances
  /* ... */
      {
        /*
         * if 'n' does not have known distance,
         * find where it occurs in 'to_be_done' and update
         */
  /* ... */
        to_be_done.erase( cfind );
        to_be_done.insert( {n,dclose+1} );
  /* ... */
}
```

(Note that we erase a record in the `to_be_done` set, and then re-insert the same key with a new value. We could have done a simple update if we had used a `map` instead of a `set`.)

The various places where you find nodes in the finished / unfinished sets are up to you to implement. You can use simple loops, or use `find_if` to find the elements matching the node numbers.

Exercise 27.5. Fill in the details of the above outline to realize Dijkstra's algorithm.

27.2 Linear algebra formulation

In this part of the project you will explore how much you make graph algorithms look like linear algebra.

27.2.1 Code preliminaries

27.2.1.1 Data structures

You need a matrix and a vector. The vector is easy:

```
class Vector {
private:
  vector<vectorvalue> values;
public:
  Vector( int n )
    : values( vector<vectorvalue>(n,infinite) ) {};
```

For the matrix, use initially a dense matrix:

```
class AdjacencyMatrix {
private:
  vector< vector<matrixvalue> > adjacency;
public:
  AdjacencyMatrix(int n)
    : adjacency( vector<vector<matrixvalue>>
                 ( n,vector<matrixvalue>(n,empty) ) ) {
  };
```

but later we will optimize that.

Remark 18 *In general it's not a good idea to store a matrix as a vector-of-vectors, but in this case we need to be able to return a matrix row, so it is convenient.*

27.2.1.2 Matrix vector multiply

Let's write a routine

```
Vector AdjacencyMatrix::leftmultiply( const Vector& left );
```

This is the simplest solution, but not necessarily the most efficient one, as it creates a new vector object for each matrix-vector multiply.

As explained in the theory background, graph algorithms can be formulated as matrix-vector multiplications with unusual add/multiply operations. Thus, the core of the multiplication routine could look like

```
for ( int row=0; row<n; ++row ) {
  for ( int col=0; col<n; ++col) {
    result[col] = add( result[col], mult( left[row],adjacency[row][col] ) );
  }
}
```

27.2.2 Unweighted graphs

Exercise 27.6. Implement the `add` / `mult` routines to make the SSSP algorithm on unweighted graphs work.

27.2.3 Dijkstra's algorithm

As an example, consider the following adjacency matrix:

```
.  1  .  .  5
.  .  1  .  .
.  .  .  1  .
.  .  .  .  1
1  .  .  .  .
```

The shortest distance $0 \to 4$ is 4, but in the first step a larger distance of 5 is discovered. Your algorithm should show an output similar to this for the successive updates to the known shortest distances:

```
Input :  0 . . . .
step 0:  0 1 . . 5
step 1:  0 1 2 . 5
step 2:  0 1 2 3 5
step 3:  0 1 2 3 4
```

Exercise 27.7. Implement new versions of the `add` / `mult` routines to make the matrix-vector multiplication correspond to Dijkstra's algorithm for SSSP on weighted graphs.

27.2.4 Sparse matrices

The matrix data structure described above can be made more compact by storing only nonzero elements. Implement this.

27.2.5 Further explorations

How elegant can you make your code through operator overloading?

Can you code the all-pairs shortest path algorithm?

Can you extend the SSSP algorithm to also generate the actual paths?

27.3 Tests and reporting

You now have two completely different implementations of some graph algorithms. Generate some large matrices and time the algorithms.

Discuss your findings, paying attention to amount of work performed and amount of memory needed.

Chapter 28

Congestion

This section contains a sequence of exercises that builds up to a simulation of car traffic.

28.1 Problem statement

Car traffic is determined by individual behaviors: drivers accelerating, braking, deciding to go one way or another. From this, emergent phenomena arise, most clearly of course traffic jams.

28.2 Code design

In this project we will explore the so-called *actor model*: the simulation consists of independent actors that react to each others' actions.

We need only two basic classes:

1. A `Car` class, where objects have a location and a speed, and they can react to the car in front of them. (For simplicity we assume that drivers only look at the car immediately ahead of them.)
2. a `Street` class, where each street is a container for a number of cars.

There are various ways we can run a simulation (with *thread*s, driven by *interrupt*s), but for simplicity we consider discrete time steps. This means that both the car and street class have a `progress` method that advanced the object by one time step.

28.2.1 Cars

The main thing you can ask of a car, is to move by a time step. This means you need to have its location, speed, and direction. It makes sense for a car to know its own speed and location, but the direction derives from the street. Thus, the street could update the cars as:

```
for ( auto c : cars() ) {
  c->progress( unit_vector() );
}
```

and the car would take the direction as an input.

28.2.2 Street

Let's start wit one-way streets. A street is then a collection of cars, but, until they start passing each other, they are actually ordered. A vector would be suitable for that. However, a car can move to a different street, or even be in two streets at once if it's on an intersection. For that reason, model a street as

```
class Street {
private:
  vector<shared_ptr<Car>> cars;
```

Once we get the simulation going, cars are going to react to each other's behavior, in particular the behavior of the car immediately in front of them. There are various ways you can model that.

1. For instance, the street can pass the information of one car to the next.
2. Possibly more elegant, each car has a pointer to the one before and after it.

28.2.3 Unit tests

At this level you can do a number of unit tests to ensure that your code behaves as intended.

1. Define a street, and test its unit vector.
2. Place a car in various locations, both on the street and beyond or the side of it. Write test methods for these cases, for instance **missing snippet streettestbeyond**
3. If you put a car on the street, its speed stays constant, and it leaves the street after a predictable amount of time:

```
REQUIRE_NOTHROW( main_street.insert_with_speed(speed) );
float time=0.f;
for ( int t=0; t<static_cast<int>(main_street.length()); ++t ) { //
    one extra step
  INFO( format( "at t={}, #cars={}",t,main_street.size() ) );
  if ( main_street.empty() ) break;

  if (global_do_vis) {
    // animation:
    main_street.display();
    std::this_thread::sleep_for( seconds{2}/10. );
    cout << '\n';
    printf( "%c[1A",(char)27); // line up again.
  }

  auto car = main_street.at(0);
  REQUIRE( car->speed()==Catch::Approx(speed) );
  REQUIRE_NOTHROW( main_street.progress() );
}
REQUIRE( main_street.empty() );
```

Chapter 29

DNA Sequencing

In this set of exercises you will write mechanisms for DNA sequencing.

29.1 Basic functions

Refer to section **??**.

First we set up some basic mechanisms.

Exercise 29.1. There are four bases, `A,C,G,T`, and each has a complement: `A` ↔ `T`, `C` ↔ `G`. Implement this through a `map`, and write a function

```
char BaseComplement(char);
```

Exercise 29.2. Write code to read a *Fasta* file into a `string`. The first line, starting with >, is a comment; all other lines should be concatenated into a single string denoting the genome.

Read the virus genome in the file `lambda_virus.fa`.

Count the four bases in the genome two different ways. First use a `map`. Time how long this takes. Then do the same thing using an array of length four, and a conditional statement.

Bonus: try to come up with a faster way of counting. Use a vector of length 4, and find a way of computing the index directly from the letters `A,C,G,T`. Hint: research ASCII codes and possibly bit operations.

29.2 De novo shotgun assembly

One approach to generating a genome is to cut it to pieces, and algorithmically glue them back together. (It is much easier to sequence the bases of a short read than of a very long genome.)

If we assume that we have enough reads that each genome position is covered, we can look at the overlaps by the reads. One heuristic is then to find the Shortest Common Superset (SCS).

29.2.1 Overlap layout consensus

1. Make a graph where the reads are the vertices, and vertices are connected if they overlap; the amount of overlap is the edge weight.
2. The SCS is then a *Hamiltonian path* through this graph – this is already NP-complete.
3. Additionally, we optimize for maximum total overlap.
   ⇒ Traveling Salesman Problem (TSP), NP-hard.

Rather than finding the optimal superset, we can use a greedy algorithm, where every time we find the read with maximal overlap.

Repeats are often a problem. Another is spurious subgraphs from sequencing errors.

29.2.2 De Bruijn graph assembly

29.3 'Read' matching

A 'read' is a short fragment of DNA, that we want to match against a genome. In this section you will explore algorithms for this type of matching.

While here we mostly consider the context of genomics, such algorithms have other applications. For instance, searching for a word in a web page is essentially the same problem. Consequently, there is a considerable history of this topic.

29.3.1 Naive matching

We first explore a naive matching algorithm: for each location in the genome, see if the read matches up.

```
ATACTGACCAAGAACGTGATTACTTCATGCAGCGTTACCAT
  ACCAAGAACGTG
    ^ mismatch

ATACTGACCAAGAACGTGATTACTTCATGCAGCGTTACCAT
      ACCAAGAACGTG
      total match
```

Exercise 29.3. Code up the naive algorithm for matching a read. Test it on fake reads obtained by copying a substring from the genome. Use the genome in `phix.fa`.

Now read the *Fastq* file `ERR266411_1.first1000.fastq`. Fastq files contains groups of four lines: the second line in each group contains the reads. How many of these reads are matched to the genome?

Reads are not necessarily a perfect match; in fact, each fourth line in the fastq file gives an indication of the 'quality' of the corresponding read. How many matches do you get if you take a substring of the first 30 or so characters of each read?

29.3.2 Boyer-Moore matching

The *Boyer-Moore* string matching algorithm [7] is much faster than naive matching, since it uses two clever tricks to weed out comparisons that would not give a match.

Bad character rule

In naive matching, we determined match locations left-to-right, and then tried matching left-to-right. In Bowers-Moore (BM), we still find match locations left-to-right, but we do our matching right-to-left.

```
   vvvv match location
antidisestablishmentarianism
   blis
    ^bad character
```

The mismatch is an ‘l’ in the pattern, which does not match a ‘d’ in the text. Since there is no ‘d’ in the pattern at all, we move the pattern completely past the mismatch:

```
   vvvv match location
antidisestablishmentarianism
       blis
```

in fact, we move it further, to the first match on the first character of the pattern:

```
           vvvv match location
antidisestablishmentarianism
           blis
           ^ first character match
```

The case where we have a mismatch, but the character in the text does appear in the pattern is a little trickier: we find the next occurrence of the mismatched character in the pattern, and use that to determine the shift distance.

```
shoobeedoobeeboobah
edoobeeboob
       ^ mismatch
 ^ other occurrence of 'd'
```

Note that this can be a considerably smaller shift than in the previous case.

```
        v
shoobeedoobeeboobah
      edoobeeboob
       ^ match the bad character 'd'
      ^ new location
```

Exercise 29.4. Discuss how efficient you expect this heuristic to be in the context of genomics versus text searching. (See above.)

Good suffix rule

The ‘good suffix’ consists of the matched characters after the bad character. When moving the read, we try to keep the good suffix intact:

```
desistrust
```

```
  listrest
        ^^good suffix

desistrust
      listrest
        ^^next occurrence of suffix
```

Chapter 30

Memory allocation

This project is not yet ready

https://www.youtube.com/watch?v=R3cBbvIFqFk

Monotonic allocator

- base and free pointer,
- always allocate from the free location
- only release when everything has been freed.

appropriate:

- video processing: release everything used for one frame
- event processing: release everything used for handling the event

Stack allocator

Chapter 31

Ballistics calculations

THIS PROJECT IS NOT READY FOR PRIME TIME

31.1 Introduction

From https://encyclopedia2.thefreedictionary.com/ballistics

> Ballistics
>
> the science of the movement of artillery shells, bullets, mortar shells, aerial bombs, rocket artillery projectiles and missiles, harpoons, and so on. Ballistics is a technical military science based on a set of physics and mathematics disciplines. Interior ballistics is distinguished from exterior ballistics.
>
> Interior ballistics is concerned with the movement of a projectile (or other body whose mechanical freedom is restricted by certain conditions) in the bore of a gun under the influence of powder gases as well as the rules governing other processes occurring in the bore or in the chamber of a powder rocket during firing. Interior ballistics views the firing as a complex process of rapid transformation of the powder's chemical energy into heat energy and then into the mechanical work of displacing the projectile, charge, and recoil parts of the gun. In interior ballistics the different periods that are distinguished in the firing are the preliminary period, which is from the start of the powder combustion until the projectile begins to move; the first (primary) period, which is from the start of projectile movement until the end of powder combustion; the second period, which is from the end of powder combustion until the moment that the projectile leaves the bore (the period of adiabatic expansion of the gases); and the period of the aftereffect of the powder gases on the projectile and barrel. The laws governing the processes related to this last period are dealt with in a special division of ballistics, known as intermediate ballistics. The end of the period of aftereffect on the projectile divides the phenomena studied by interior and exterior ballistics.
>
> The main divisions of interior ballistics are "pyrostatics," "pyrodynamics," and ballistic gun design. Pyrostatics is the study of the laws of powder combustion and gas formation during the combustion of powder in a constant volume in which the effect of the chemical composition of the powder and its forms and dimensions on the laws of combustion and gas formation is determined. Pyrodynamics is concerned

with the study of the processes and phenomena that take place in the bore during firing and the determination of the relationships between the design features of the bore, the conditions of loading, and various physical-chemical and mechanical processes that occur during firing. On the basis of a consideration of these processes and also of the forces operating on the projectile and barrel, a system of equations is established that describes the firing process, including the basic equation of interior ballistics, which relates the magnitude of the burned part of the charge, the pressure of powder gases in the bore, the velocity of the projectile, and the length of the path it has traveled. The solution of this system and the discovery of the relationship between change in the pressure ρ of the powder gases, the velocity v of the projectile, and other parameters on path l of the projectile and the time it has moved along the bore is the first main (direct)

to solve this problem the analytic method, numerical integration methods (including those based on computers), and tabular methods are used. In view of the complexity of the firing process and insufficient study of particular factors, certain assumptions are made. The correction formulas of interior ballistics are of great practical significance; they make it possible to determine the change in muzzle velocity of the projectile and maximum pressure in the bore when there are changes made in the loading conditions.

Ballistic gun design is the second main (correlative) problem of interior ballistics. By it are determined the design specifications of the bore and the loading conditions under which a projectile of given caliber and mass will obtain an assigned (muzzle) velocity in flight. The curves of change in the pressure of the gases in the bore and of the velocity of the projectile along the length of the barrel and in time are calculated for the variation of the barrel selected during designing. These curves are the initial data in designing the artillery system as a whole and the ammunition for it. Internal ballistics also includes the study of the firing process in the rifle, in cases when special and combined charges are used, in systems with conical barrels, and in systems in which gases are exhausted during powder combustion (high-low pressure guns and recoilless guns, infantry mortars). Another important division is the interior ballistics of powder rockets, which has developed into a special science. The main divisions of the interior ballistics of powder rockets are pyrostatics of a semiclosed space, which consider the laws of powder combustion at comparatively low and constant pressure; the solution of the basic problem of the interior ballistics of powder rockets, which is to determine (under set loading conditions) the rules of variation in pressure of the powder gases in the chamber with regard to time and to determine the rules of variation in thrust necessary to ensure the required rocket velocity; the ballistic design of powder rockets, which involves determining the energy-producing characteristics of the powder, the weight and form of the charge, and the design parameters of the nozzle which ensure, with an assigned weight of the rocket's warhead, the necessary thrust force during its operation.

Exterior ballistics is concerned with the study of the movement of unguided projectiles (mortar shells, bullets, and so on) after they leave the bore (or launcher) and the factors that affect this movement. It includes basically the study of all the elements of motion of the projectile and of the forces that act upon it in flight (the force of air resistance, the force of gravity, reactive force, the force arising in the

aftereffect period, and so on); the study of the movement of the center of mass of the projectile for the purpose of calculating its trajectory (see Figure 2) when there are set initial and external conditions (the basic problem of exterior ballistics); and the determination of the flight stability and dispersion of projectiles. Two important divisions of exterior ballistics are the theory of corrections, which develops methods of evaluating the influence of the factors that determine the projectile's flight on the nature of its trajectory, and the technique for drawing up firing tables and of finding the optimal exterior ballistics variation in the designing of artillery systems. The theoretical solution of the problems of projectile movement and of the problems of the theory of corrections amounts to making up equations for the projectile's movement, simplifying these equations, and seeking methods of solving them. This has been made significantly easier and faster with the appearance of computers. In order to determine the initial conditions—that is, initial velocity and angle of departure, shape and mass of the projectile—which are necessary to obtain a given trajectory, special tables are used in exterior ballistics. The working out of the technique for drawing up a firing table involves determining the optimal combination of theoretical and experimental research that makes it possible to obtain firing tables of the required accuracy with the minimal expenditure of time. The methods of exterior ballistics are also used in the study of the laws of movement of spacecraft (during their movement without the influence of controlling forces and moments). With the appearance of guided missiles, exterior ballistics played a major part in the formation and development of the theory of flight and became a particular instance of this theory.

The appearance of ballistics as a science dates to the 16th century. The first works on ballistics are the books by the Italian N. Tartaglia, A New Science (1537) and Questions

and Discoveries Relating to Artillery Fire (1546). In the 17th century the fundamental principles of exterior ballistics were established by Galileo, who developed the parabolic theory of projectile movement, by the Italian E. Torricelli, and by the Frenchman M. Mersenne, who proposed that the science of the movement of projectiles be called ballistics (1644). I. Newton made the first investigations of the movement of a projectile, taking air resistance into consideration (Mathematical Principles of Natural Philosophy,1687). During the 17th and 18th centuries the movement of projectiles was studied by the Dutchman C. Huygens, the Frenchman P. Varignon, the Englishman B. Robins, the Swiss D. Bernoulli, the Russian scientist L. Eiler, and others. The experimental and theoretical foundations of interior ballistics were laid in the 18th century in works of Robins, C. Hutton, Bernoulli, and others. In the 19th century the laws of air resistance were established (the laws of N. V. Maievskii and N. A. Zabudskii, Havre's law, and A. F. Siacci's law).

The numerical analysis of ballistics calculations on the *ENIAC* are described in [19].

31.1.1 Physics

These are the governing equations:

$$\begin{aligned} x'' &= -E(x' - w_x) + 2\Omega \cos L \sin \alpha y' \\ y'' &= -Ey' - g - 2\Omega \cos L \sin \alpha x' \\ z'' &= -E(z' - w_z) + 2\Omega \sin L x' + 2\Omega \cos L \cos \alpha y' \end{aligned} \tag{31.1}$$

where

- x, y, z are the quantities of interest: distance, altitude, sideways displacement;
- w_x, w_z are wind speed;
- E is a complicated function of y, involving air density and speed of sound;
- α is the azimuth, that is, angle of firing;
- All other quantities are needed for physical realism, but will be set $\equiv 1$ in this coding exercise.

31.1.2 Numerical analysis

This uses an Euler-MacLaurin scheme of third order:

$$f_1 - f_0 = \frac{1}{2}(f'_0 + f'_1)h + \frac{1}{12}(f'_0 - f'_1)h^2 + O(h^5) \tag{31.2}$$

which works out to

$$\begin{aligned} \bar{x}'_1 &= x'_0 + x''_0 \Delta t \\ x_1 &= x_0 + x'_0 \Delta t \\ x'_1 &= x'_0 + (x''_0 + \bar{x}''_1)\tfrac{\Delta t}{2} \\ x_1 &= x_0 (x'_0 + \bar{x}'_1)\tfrac{\Delta t}{2} + (x''_0 - \bar{x}''_1)\tfrac{\Delta t^2}{12} \end{aligned} \tag{31.3}$$

Chapter 32

Cryptography

32.1 The basics

While floating point numbers can span a rather large range – up to 10^{300} or so for double precision – integers have a much smaller one: up to about 10^9. That is not enough to do cryptographic applications, which deal in much larger numbers. (Why can't we use floating point numbers?)

So the first step is to write classes `Integer` and `Fraction` that have no such limitations. Use operator overloading to make simple expressions work:

```
Integer big=2000000000; // two billion
big *= 1000000; bigger = big+1;
Integer one = bigger % big;
```

Exercise 32.1. Code Farey sequences.

32.2 Cryptography

https://simple.wikipedia.org/wiki/RSA_algorithm

https://simple.wikipedia.org/wiki/Exponentiation_by_squaring

32.3 Blockchain

Implement a blockchain algorithm.

PART III

CARPENTRY

Chapter 33

Build systems

33.1 The Unix 'Make' utility

The *Make* utility helps you manage the building of projects: its main task is to facilitate rebuilding only those parts of a multi-file project that need to be recompiled or rebuilt. This can save lots of time, since it can replace a minutes-long full installation by a single file compilation. *Make* can also help maintaining multiple installations of a program on a single machine, for instance compiling a library with more than one compiler, or compiling a program in debug and optimized mode.

Make is a Unix utility with a long history, and traditionally there are variants with slightly different behavior, for instance on the various flavors of Unix such as HP-UX, AUX, IRIX. These days, it is advisable, no matter the platform, to use the GNU version of Make which has some very powerful extensions; it is available on all Unix platforms (on Linux it is the only available variant), and it is a *de facto* standard. The manual is available at http://www.gnu.org/software/make/manual/make.html, or you can read the book [18].

There are other build systems, most notably *Scons* and *Bjam*. We will not discuss those here. The examples in this tutorial will be for the C and Fortran languages, but *Make* can work with any language, and in fact with things like TEX that are not really a language at all; see section 33.8.

33.2 A simple example

Purpose. In this section you will see a simple example, just to give the flavor of *Make*.

The files for this section can be found in the repository.

33.2.1 C++

Make the following files:

foo.cxx

```
#include "papi_test.h"
extern int TESTS_QUIET;         /* Declared in test_utils.c */

#define PCHECK(e) if (e!=PAPI_OK) \
        {printf("Problem in papi call, line %d\n",__LINE__); \
```

```
         return 1;}
#define NEVENTS 2
/* we are assuming array storage in Fortran column mode */
#define INDEX(i,j,m,n) i+j*m

double *array;

void clear_right(int m,int n) {
  int i,j;
  for (j=0; j<n; j++)
    for (i=0; i<m; i++)
      array[INDEX(i,j,m,n)] = 0;
  return;
}

void clear_wrong(int m,int n) {
  int i,j;
  for (i=0; i<m; i++)
    for (j=0; j<n; j++)
      array[INDEX(i,j,m,n)] = 0;
  return;
}

void do_operation_right(int m,int n) {
  int i,j;
  for (j=0; j<n; j++)
    for (i=0; i<m; i++)
      array[INDEX(i,j,m,n)] = array[INDEX(i,j,m,n)]+1;
  return;
}

void do_operation_wrong(int m,int n) {
  int i,j;
  for (i=0; i<m; i++)
    for (j=0; j<n; j++)
      array[INDEX(i,j,m,n)] = array[INDEX(i,j,m,n)]+1;
  return;
}

#define COL 1
#define ROW 2
int main(int argc, char **argv)
{
   int events[NEVENTS] = {PAPI_TLB_DM,PAPI_TOT_CYC};
   long_long values[NEVENTS];
   int retval,order=COL;
   PAPI_event_info_t info, info1;
   const PAPI_hw_info_t *hwinfo = NULL;
   int event_code;
   const PAPI_substrate_info_t *s = NULL;

   tests_quiet(argc, argv);     /* Set TESTS_QUIET variable */
   if (argc==2 && !strcmp(argv[1],"row")) {
     printf("wrong way\n"); order=ROW;
   } else printf("right way\n");

   retval = PAPI_library_init(PAPI_VER_CURRENT);
```

```
   if (retval != PAPI_VER_CURRENT)
      test_fail(__FILE__, __LINE__, "PAPI_library_init", retval);

   {
     int i;
     for (i=0; i<NEVENTS; i++) {
       retval = PAPI_query_event(events[i]); PCHECK(retval);
     }
   }

#define M 1000
#define N 2000
   {
     int m,n;
     m = M;
     array = (double*) malloc(M*N*sizeof(double));
     for (n=10; n<N; n+=10) {
       if (order==COL)
 clear_right(m,n);
       else
 clear_wrong(m,n);
       retval = PAPI_start_counters(events,NEVENTS); PCHECK(retval);
       if (order==COL)
 do_operation_right(m,n);
       else
 do_operation_wrong(m,n);
       retval = PAPI_stop_counters(values,NEVENTS); PCHECK(retval);
       printf("m,n=%d,%d\n#elements:\t%d\n",m,n,m*n);
       printf("Tot cycles: %d\nTLB misses:\t%d\nmisses per column:\t%9.5f\n\n",
      values[1],values[0],values[0]/(1.*n));
     }
     free(array);
   }

   return 0;
}
```

bar.cxx

```
#include "papi_test.h"
extern int TESTS_QUIET;         /* Declared in test_utils.c */

#define PCHECK(e) if (e!=PAPI_OK) \
        {printf("Problem in papi call, line %d\n",__LINE__); \
         return 1;}
#define NEVENTS 2
/* we are assuming array storage in Fortran column mode */
#define INDEX(i,j,m,n) i+j*m

double *array;

void clear_right(int m,int n) {
  int i,j;
  for (j=0; j<n; j++)
    for (i=0; i<m; i++)
      array[INDEX(i,j,m,n)] = 0;
  return;
```

```
}

void clear_wrong(int m,int n) {
  int i,j;
  for (i=0; i<m; i++)
    for (j=0; j<n; j++)
      array[INDEX(i,j,m,n)] = 0;
  return;
}

void do_operation_right(int m,int n) {
  int i,j;
  for (j=0; j<n; j++)
    for (i=0; i<m; i++)
      array[INDEX(i,j,m,n)] = array[INDEX(i,j,m,n)]+1;
  return;
}

void do_operation_wrong(int m,int n) {
  int i,j;
  for (i=0; i<m; i++)
    for (j=0; j<n; j++)
      array[INDEX(i,j,m,n)] = array[INDEX(i,j,m,n)]+1;
  return;
}

#define COL 1
#define ROW 2
int main(int argc, char **argv)
{
   int events[NEVENTS] = {PAPI_TLB_DM,PAPI_TOT_CYC};
   long_long values[NEVENTS];
   int retval,order=COL;
   PAPI_event_info_t info, info1;
   const PAPI_hw_info_t *hwinfo = NULL;
   int event_code;
   const PAPI_substrate_info_t *s = NULL;

   tests_quiet(argc, argv);     /* Set TESTS_QUIET variable */
   if (argc==2 && !strcmp(argv[1],"row")) {
     printf("wrong way\n"); order=ROW;
   } else printf("right way\n");

   retval = PAPI_library_init(PAPI_VER_CURRENT);
   if (retval != PAPI_VER_CURRENT)
      test_fail(__FILE__, __LINE__, "PAPI_library_init", retval);

   {
     int i;
     for (i=0; i<NEVENTS; i++) {
       retval = PAPI_query_event(events[i]); PCHECK(retval);
     }
   }

#define M 1000
#define N 2000
   {
```

```
      int m,n;
      m = M;
      array = (double*) malloc(M*N*sizeof(double));
      for (n=10; n<N; n+=10) {
        if (order==COL)
 clear_right(m,n);
        else
 clear_wrong(m,n);
        retval = PAPI_start_counters(events,NEVENTS); PCHECK(retval);
        if (order==COL)
 do_operation_right(m,n);
        else
 do_operation_wrong(m,n);
        retval = PAPI_stop_counters(values,NEVENTS); PCHECK(retval);
        printf("m,n=%d,%d\n#elements:\t%d\n",m,n,m*n);
        printf("Tot cycles: %d\nTLB misses:\t%d\nmisses per column:\t%9.5f\n\n",
       values[1],values[0],values[0]/(1.*n));
      }
      free(array);
    }

    return 0;
}
```

bar.h

```
#include "papi_test.h"
extern int TESTS_QUIET;         /* Declared in test_utils.c */

#define PCHECK(e) if (e!=PAPI_OK) \
        {printf("Problem in papi call, line %d\n",__LINE__); \
         return 1;}
#define NEVENTS 2
/* we are assuming array storage in Fortran column mode */
#define INDEX(i,j,m,n) i+j*m

double *array;

void clear_right(int m,int n) {
  int i,j;
  for (j=0; j<n; j++)
    for (i=0; i<m; i++)
      array[INDEX(i,j,m,n)] = 0;
  return;
}

void clear_wrong(int m,int n) {
  int i,j;
  for (i=0; i<m; i++)
    for (j=0; j<n; j++)
      array[INDEX(i,j,m,n)] = 0;
  return;
}

void do_operation_right(int m,int n) {
  int i,j;
  for (j=0; j<n; j++)
```

```
    for (i=0; i<m; i++)
      array[INDEX(i,j,m,n)] = array[INDEX(i,j,m,n)]+1;
  return;
}

void do_operation_wrong(int m,int n) {
  int i,j;
  for (i=0; i<m; i++)
    for (j=0; j<n; j++)
      array[INDEX(i,j,m,n)] = array[INDEX(i,j,m,n)]+1;
  return;
}

#define COL 1
#define ROW 2
int main(int argc, char **argv)
{
   int events[NEVENTS] = {PAPI_TLB_DM,PAPI_TOT_CYC};
   long_long values[NEVENTS];
   int retval,order=COL;
   PAPI_event_info_t info, info1;
   const PAPI_hw_info_t *hwinfo = NULL;
   int event_code;
   const PAPI_substrate_info_t *s = NULL;

   tests_quiet(argc, argv);     /* Set TESTS_QUIET variable */
   if (argc==2 && !strcmp(argv[1],"row")) {
     printf("wrong way\n"); order=ROW;
   } else printf("right way\n");

   retval = PAPI_library_init(PAPI_VER_CURRENT);
   if (retval != PAPI_VER_CURRENT)
      test_fail(__FILE__, __LINE__, "PAPI_library_init", retval);

   {
     int i;
     for (i=0; i<NEVENTS; i++) {
       retval = PAPI_query_event(events[i]); PCHECK(retval);
     }
   }

#define M 1000
#define N 2000
   {
     int m,n;
     m = M;
     array = (double*) malloc(M*N*sizeof(double));
     for (n=10; n<N; n+=10) {
       if (order==COL)
 clear_right(m,n);
       else
 clear_wrong(m,n);
       retval = PAPI_start_counters(events,NEVENTS); PCHECK(retval);
       if (order==COL)
 do_operation_right(m,n);
       else
 do_operation_wrong(m,n);
```

```
        retval = PAPI_stop_counters(values,NEVENTS); PCHECK(retval);
        printf("m,n=%d,%d\n#elements:\t%d\n",m,n,m*n);
        printf("Tot cycles: %d\nTLB misses:\t%d\nmisses per column:\t%9.5f\n\n",
       values[1],values[0],values[0]/(1.*n));
      }
      free(array);
    }

    return 0;
}
```

and a makefile:

Makefile

```
#include "papi_test.h"
extern int TESTS_QUIET;         /* Declared in test_utils.c */

#define PCHECK(e) if (e!=PAPI_OK) \
        {printf("Problem in papi call, line %d\n",__LINE__); \
         return 1;}
#define NEVENTS 2
/* we are assuming array storage in Fortran column mode */
#define INDEX(i,j,m,n) i+j*m

double *array;

void clear_right(int m,int n) {
  int i,j;
  for (j=0; j<n; j++)
    for (i=0; i<m; i++)
      array[INDEX(i,j,m,n)] = 0;
  return;
}

void clear_wrong(int m,int n) {
  int i,j;
  for (i=0; i<m; i++)
    for (j=0; j<n; j++)
      array[INDEX(i,j,m,n)] = 0;
  return;
}

void do_operation_right(int m,int n) {
  int i,j;
  for (j=0; j<n; j++)
    for (i=0; i<m; i++)
      array[INDEX(i,j,m,n)] = array[INDEX(i,j,m,n)]+1;
  return;
}

void do_operation_wrong(int m,int n) {
  int i,j;
  for (i=0; i<m; i++)
    for (j=0; j<n; j++)
      array[INDEX(i,j,m,n)] = array[INDEX(i,j,m,n)]+1;
  return;
```

```
}

#define COL 1
#define ROW 2
int main(int argc, char **argv)
{
   int events[NEVENTS] = {PAPI_TLB_DM,PAPI_TOT_CYC};
   long_long values[NEVENTS];
   int retval,order=COL;
   PAPI_event_info_t info, info1;
   const PAPI_hw_info_t *hwinfo = NULL;
   int event_code;
   const PAPI_substrate_info_t *s = NULL;

   tests_quiet(argc, argv);     /* Set TESTS_QUIET variable */
   if (argc==2 && !strcmp(argv[1],"row")) {
     printf("wrong way\n"); order=ROW;
   } else printf("right way\n");

   retval = PAPI_library_init(PAPI_VER_CURRENT);
   if (retval != PAPI_VER_CURRENT)
      test_fail(__FILE__, __LINE__, "PAPI_library_init", retval);

   {
     int i;
     for (i=0; i<NEVENTS; i++) {
       retval = PAPI_query_event(events[i]); PCHECK(retval);
     }
   }

#define M 1000
#define N 2000
   {
     int m,n;
     m = M;
     array = (double*) malloc(M*N*sizeof(double));
     for (n=10; n<N; n+=10) {
       if (order==COL)
 clear_right(m,n);
       else
 clear_wrong(m,n);
       retval = PAPI_start_counters(events,NEVENTS); PCHECK(retval);
       if (order==COL)
 do_operation_right(m,n);
       else
 do_operation_wrong(m,n);
       retval = PAPI_stop_counters(values,NEVENTS); PCHECK(retval);
       printf("m,n=%d,%d\n#elements:\t%d\n",m,n,m*n);
       printf("Tot cycles: %d\nTLB misses:\t%d\nmisses per column:\t%9.5f\n\n",
      values[1],values[0],values[0]/(1.*n));
     }
     free(array);
   }

   return 0;
}
```

The makefile has a number of rules like

```
foo.o : foo.cxx
<TAB>icpc -c foo.cxx
```

which have the general form

```
target : prerequisite(s)
<TAB>rule(s)
```

where the rule lines are indented by a `TAB` character.

A rule, such as above, states that a 'target' file `foo.o` is made from a 'prerequisite' `foo.cxx`, namely by executing the command `icpc -c foo.cxx`. (Here we are using the *Inte C++ compiler* `icpc`; your system could have a different compiler, such as `clang++` or `g++`.)

The precise definition of the rule is:

- if the target `foo.o` does not exist or is older than the prerequisite `foo.cxx`,
- then the command part of the rule is executed: `icpc -c foo.cxx`
- If the prerequisite is itself the target of another rule, than that rule is executed first.

Exercise. Call `make`.

> *Expected outcome.* The above rules are applied: `make` without arguments tries to build the first target, `fooprog`. In order to build this, it needs the prerequisites `foo.o` and `bar.o`, which do not exist. However, there are rules for making them, which `make` recursively invokes. Hence you see two compilations, for `foo.o` and `bar.o`, and a link command for `fooprog`.
>
> *Caveats.* Typos in the makefile or in file names can cause various errors. In particular, make sure you use tabs and not spaces for the rule lines. Unfortunately, debugging a makefile is not simple. *Make*'s error message will usually give you the line number in the make file where the error was detected.

Exercise. Do `make clean`, followed by `mv foo.cxx boo.cxx` and `make` again. Explain the error message. Restore the original file name.

> *Expected outcome. Make* will complain that there is no rule to make `foo.cxx`. This error was caused when `foo.cxx` was a prerequisite for making `foo.o`, and was found not to exist. *Make* then went looking for a rule to make it and no rule for creating `.cxx` files exists.

Now add a second argument to the function `bar`. This would require you to edit all of `bar.cxx`, `bar.h`, and `foo.cxx`, but let's say we forget to edit the last two, so only edit `bar.cxx` However, it also requires you to edit `foo.c`, but let us for now 'forget' to do that. We will see how *Make* can help you find the resulting error.

Exercise. Call `make` to recompile your program. Did it recompile `foo.c`?

> *Expected outcome.* You see that inclusion of the 'wrong' header file does not lead to an error, because C++ has polymorphism. In C this would definitely give an error. The error only shows up in the linker stage because of an unresolved reference.

Exercise. Update the header file, and call `make` again. What happens, and what had you been hoping would happen?

Expected outcome. Only the linker stage is done, and it gives the same error about an unresolved reference. Were you hoping that the main program would be recompiled?

Caveats.

The way out of this problem is to tie the header file to the source files in the makefile.

In the makefile, change the line

```
foo.o : foo.cxx
```

to

```
foo.o : foo.cxx bar.h
```

which adds `bar.h` as a prerequisite for `foo.o`. This means that, in this case where `foo.o` already exists, *Make* will check that `foo.o` is not older than any of its prerequisites. Since `bar.h` has been edited, it is younger than `foo.o`, so `foo.o` needs to be reconstructed.

Remark 19 *As already noted above, in C++ fewer errors are caught by this mechanism than in C, because of polymorphism. You might wonder if it would be possible to generate header files automatically. This is of course possible with suitable shell scripts, but tools such as* `Make` *(or* `CMake`*) do not have this built in.*

33.2.2 C

Make the following files:

`foo.c`

```
#include "papi_test.h"
extern int TESTS_QUIET;         /* Declared in test_utils.c */

#define PCHECK(e) if (e!=PAPI_OK) \
        {printf("Problem in papi call, line %d\n",__LINE__); \
         return 1;}
#define NEVENTS 2
/* we are assuming array storage in Fortran column mode */
#define INDEX(i,j,m,n) i+j*m

double *array;

void clear_right(int m,int n) {
  int i,j;
  for (j=0; j<n; j++)
    for (i=0; i<m; i++)
      array[INDEX(i,j,m,n)] = 0;
  return;
}

void clear_wrong(int m,int n) {
  int i,j;
  for (i=0; i<m; i++)
    for (j=0; j<n; j++)
      array[INDEX(i,j,m,n)] = 0;
  return;
}
```

```
void do_operation_right(int m,int n) {
  int i,j;
  for (j=0; j<n; j++)
    for (i=0; i<m; i++)
      array[INDEX(i,j,m,n)] = array[INDEX(i,j,m,n)]+1;
  return;
}

void do_operation_wrong(int m,int n) {
  int i,j;
  for (i=0; i<m; i++)
    for (j=0; j<n; j++)
      array[INDEX(i,j,m,n)] = array[INDEX(i,j,m,n)]+1;
  return;
}

#define COL 1
#define ROW 2
int main(int argc, char **argv)
{
   int events[NEVENTS] = {PAPI_TLB_DM,PAPI_TOT_CYC};
   long_long values[NEVENTS];
   int retval,order=COL;
   PAPI_event_info_t info, info1;
   const PAPI_hw_info_t *hwinfo = NULL;
   int event_code;
   const PAPI_substrate_info_t *s = NULL;

   tests_quiet(argc, argv);     /* Set TESTS_QUIET variable */
   if (argc==2 && !strcmp(argv[1],"row")) {
     printf("wrong way\n"); order=ROW;
   } else printf("right way\n");

   retval = PAPI_library_init(PAPI_VER_CURRENT);
   if (retval != PAPI_VER_CURRENT)
      test_fail(__FILE__, __LINE__, "PAPI_library_init", retval);

   {
     int i;
     for (i=0; i<NEVENTS; i++) {
       retval = PAPI_query_event(events[i]); PCHECK(retval);
     }
   }

#define M 1000
#define N 2000
   {
     int m,n;
     m = M;
     array = (double*) malloc(M*N*sizeof(double));
     for (n=10; n<N; n+=10) {
       if (order==COL)
 clear_right(m,n);
       else
 clear_wrong(m,n);
       retval = PAPI_start_counters(events,NEVENTS); PCHECK(retval);
```

```
      if (order==COL)
 do_operation_right(m,n);
      else
 do_operation_wrong(m,n);
      retval = PAPI_stop_counters(values,NEVENTS); PCHECK(retval);
      printf("m,n=%d,%d\n#elements:\t%d\n",m,n,m*n);
      printf("Tot cycles: %d\nTLB misses:\t%d\nmisses per column:\t%9.5f\n\n",
     values[1],values[0],values[0]/(1.*n));
    }
    free(array);
  }

  return 0;
}
```

bar.c

```
#include "papi_test.h"
extern int TESTS_QUIET;         /* Declared in test_utils.c */

#define PCHECK(e) if (e!=PAPI_OK) \
        {printf("Problem in papi call, line %d\n",__LINE__); \
         return 1;}
#define NEVENTS 2
/* we are assuming array storage in Fortran column mode */
#define INDEX(i,j,m,n) i+j*m

double *array;

void clear_right(int m,int n) {
  int i,j;
  for (j=0; j<n; j++)
    for (i=0; i<m; i++)
      array[INDEX(i,j,m,n)] = 0;
  return;
}

void clear_wrong(int m,int n) {
  int i,j;
  for (i=0; i<m; i++)
    for (j=0; j<n; j++)
      array[INDEX(i,j,m,n)] = 0;
  return;
}

void do_operation_right(int m,int n) {
  int i,j;
  for (j=0; j<n; j++)
    for (i=0; i<m; i++)
      array[INDEX(i,j,m,n)] = array[INDEX(i,j,m,n)]+1;
  return;
}

void do_operation_wrong(int m,int n) {
  int i,j;
  for (i=0; i<m; i++)
    for (j=0; j<n; j++)
```

```
      array[INDEX(i,j,m,n)] = array[INDEX(i,j,m,n)]+1;
  return;
}

#define COL 1
#define ROW 2
int main(int argc, char **argv)
{
   int events[NEVENTS] = {PAPI_TLB_DM,PAPI_TOT_CYC};
   long_long values[NEVENTS];
   int retval,order=COL;
   PAPI_event_info_t info, info1;
   const PAPI_hw_info_t *hwinfo = NULL;
   int event_code;
   const PAPI_substrate_info_t *s = NULL;

   tests_quiet(argc, argv);     /* Set TESTS_QUIET variable */
   if (argc==2 && !strcmp(argv[1],"row")) {
     printf("wrong way\n"); order=ROW;
   } else printf("right way\n");

   retval = PAPI_library_init(PAPI_VER_CURRENT);
   if (retval != PAPI_VER_CURRENT)
      test_fail(__FILE__, __LINE__, "PAPI_library_init", retval);

   {
     int i;
     for (i=0; i<NEVENTS; i++) {
       retval = PAPI_query_event(events[i]); PCHECK(retval);
     }
   }

#define M 1000
#define N 2000
   {
     int m,n;
     m = M;
     array = (double*) malloc(M*N*sizeof(double));
     for (n=10; n<N; n+=10) {
       if (order==COL)
 clear_right(m,n);
       else
 clear_wrong(m,n);
       retval = PAPI_start_counters(events,NEVENTS); PCHECK(retval);
       if (order==COL)
 do_operation_right(m,n);
       else
 do_operation_wrong(m,n);
       retval = PAPI_stop_counters(values,NEVENTS); PCHECK(retval);
       printf("m,n=%d,%d\n#elements:\t%d\n",m,n,m*n);
       printf("Tot cycles: %d\nTLB misses:\t%d\nmisses per column:\t%9.5f\n\n",
      values[1],values[0],values[0]/(1.*n));
     }
     free(array);
   }

   return 0;
```

```
}
```

bar.h

```
#include "papi_test.h"
extern int TESTS_QUIET;         /* Declared in test_utils.c */

#define PCHECK(e) if (e!=PAPI_OK) \
        {printf("Problem in papi call, line %d\n",__LINE__); \
         return 1;}
#define NEVENTS 2
/* we are assuming array storage in Fortran column mode */
#define INDEX(i,j,m,n) i+j*m

double *array;

void clear_right(int m,int n) {
  int i,j;
  for (j=0; j<n; j++)
    for (i=0; i<m; i++)
      array[INDEX(i,j,m,n)] = 0;
  return;
}

void clear_wrong(int m,int n) {
  int i,j;
  for (i=0; i<m; i++)
    for (j=0; j<n; j++)
      array[INDEX(i,j,m,n)] = 0;
  return;
}

void do_operation_right(int m,int n) {
  int i,j;
  for (j=0; j<n; j++)
    for (i=0; i<m; i++)
      array[INDEX(i,j,m,n)] = array[INDEX(i,j,m,n)]+1;
  return;
}

void do_operation_wrong(int m,int n) {
  int i,j;
  for (i=0; i<m; i++)
    for (j=0; j<n; j++)
      array[INDEX(i,j,m,n)] = array[INDEX(i,j,m,n)]+1;
  return;
}

#define COL 1
#define ROW 2
int main(int argc, char **argv)
{
   int events[NEVENTS] = {PAPI_TLB_DM,PAPI_TOT_CYC};
   long_long values[NEVENTS];
   int retval,order=COL;
   PAPI_event_info_t info, info1;
   const PAPI_hw_info_t *hwinfo = NULL;
```

```
   int event_code;
   const PAPI_substrate_info_t *s = NULL;

   tests_quiet(argc, argv);     /* Set TESTS_QUIET variable */
   if (argc==2 && !strcmp(argv[1],"row")) {
     printf("wrong way\n"); order=ROW;
   } else printf("right way\n");

   retval = PAPI_library_init(PAPI_VER_CURRENT);
   if (retval != PAPI_VER_CURRENT)
      test_fail(__FILE__, __LINE__, "PAPI_library_init", retval);

   {
     int i;
     for (i=0; i<NEVENTS; i++) {
       retval = PAPI_query_event(events[i]); PCHECK(retval);
     }
   }

#define M 1000
#define N 2000
   {
     int m,n;
     m = M;
     array = (double*) malloc(M*N*sizeof(double));
     for (n=10; n<N; n+=10) {
       if (order==COL)
 clear_right(m,n);
       else
 clear_wrong(m,n);
       retval = PAPI_start_counters(events,NEVENTS); PCHECK(retval);
       if (order==COL)
 do_operation_right(m,n);
       else
 do_operation_wrong(m,n);
       retval = PAPI_stop_counters(values,NEVENTS); PCHECK(retval);
       printf("m,n=%d,%d\n#elements:\t%d\n",m,n,m*n);
       printf("Tot cycles: %d\nTLB misses:\t%d\nmisses per column:\t%9.5f\n\n",
      values[1],values[0],values[0]/(1.*n));
     }
     free(array);
   }

   return 0;
}
```

and a makefile:

`Makefile`

```
#include "papi_test.h"
extern int TESTS_QUIET;         /* Declared in test_utils.c */

#define PCHECK(e) if (e!=PAPI_OK) \
        {printf("Problem in papi call, line %d\n",__LINE__); \
         return 1;}
#define NEVENTS 2
```

```
/* we are assuming array storage in Fortran column mode */
#define INDEX(i,j,m,n) i+j*m

double *array;

void clear_right(int m,int n) {
  int i,j;
  for (j=0; j<n; j++)
    for (i=0; i<m; i++)
      array[INDEX(i,j,m,n)] = 0;
  return;
}

void clear_wrong(int m,int n) {
  int i,j;
  for (i=0; i<m; i++)
    for (j=0; j<n; j++)
      array[INDEX(i,j,m,n)] = 0;
  return;
}

void do_operation_right(int m,int n) {
  int i,j;
  for (j=0; j<n; j++)
    for (i=0; i<m; i++)
      array[INDEX(i,j,m,n)] = array[INDEX(i,j,m,n)]+1;
  return;
}

void do_operation_wrong(int m,int n) {
  int i,j;
  for (i=0; i<m; i++)
    for (j=0; j<n; j++)
      array[INDEX(i,j,m,n)] = array[INDEX(i,j,m,n)]+1;
  return;
}

#define COL 1
#define ROW 2
int main(int argc, char **argv)
{
   int events[NEVENTS] = {PAPI_TLB_DM,PAPI_TOT_CYC};
   long_long values[NEVENTS];
   int retval,order=COL;
   PAPI_event_info_t info, info1;
   const PAPI_hw_info_t *hwinfo = NULL;
   int event_code;
   const PAPI_substrate_info_t *s = NULL;

   tests_quiet(argc, argv);     /* Set TESTS_QUIET variable */
   if (argc==2 && !strcmp(argv[1],"row")) {
     printf("wrong way\n"); order=ROW;
   } else printf("right way\n");

   retval = PAPI_library_init(PAPI_VER_CURRENT);
   if (retval != PAPI_VER_CURRENT)
      test_fail(__FILE__, __LINE__, "PAPI_library_init", retval);
```

```
    {
      int i;
      for (i=0; i<NEVENTS; i++) {
        retval = PAPI_query_event(events[i]); PCHECK(retval);
      }
    }

#define M 1000
#define N 2000
    {
      int m,n;
      m = M;
      array = (double*) malloc(M*N*sizeof(double));
      for (n=10; n<N; n+=10) {
        if (order==COL)
 clear_right(m,n);
        else
 clear_wrong(m,n);
        retval = PAPI_start_counters(events,NEVENTS); PCHECK(retval);
        if (order==COL)
 do_operation_right(m,n);
        else
 do_operation_wrong(m,n);
        retval = PAPI_stop_counters(values,NEVENTS); PCHECK(retval);
        printf("m,n=%d,%d\n#elements:\t%d\n",m,n,m*n);
        printf("Tot cycles: %d\nTLB misses:\t%d\nmisses per column:\t%9.5f\n\n",
       values[1],values[0],values[0]/(1.*n));
      }
      free(array);
    }

    return 0;
}
```

The makefile has a number of rules like

```
foo.o : foo.c
<TAB>cc -c foo.c
```

which have the general form

```
target : prerequisite(s)
<TAB>rule(s)
```

where the rule lines are indented by a `TAB` character.

A rule, such as above, states that a 'target' file `foo.o` is made from a 'prerequisite' `foo.c`, namely by executing the command `cc -c foo.c`. The precise definition of the rule is:

- if the target `foo.o` does not exist or is older than the prerequisite `foo.c`,
- then the command part of the rule is executed: `cc -c foo.c`
- If the prerequisite is itself the target of another rule, than that rule is executed first.

Exercise. Call `make`.

Expected outcome. The above rules are applied: `make` without arguments tries to build the first target, `fooprog`. In order to build this, it needs the prerequisites `foo.o` and `bar.o`, which do not exist. However, there are rules for making them, which `make` recursively invokes. Hence you see two compilations, for `foo.o` and `bar.o`, and a link command for `fooprog`.

Caveats. Typos in the makefile or in file names can cause various errors. In particular, make sure you use tabs and not spaces for the rule lines. Unfortunately, debugging a makefile is not simple. *Make*'s error message will usually give you the line number in the make file where the error was detected.

Exercise. Do `make clean`, followed by `mv foo.c boo.c` and `make` again. Explain the error message. Restore the original file name.

Expected outcome. Make will complain that there is no rule to make `foo.c`. This error was caused when `foo.c` was a prerequisite for making `foo.o`, and was found not to exist. *Make* then went looking for a rule to make it and no rule for creating `.c` files exists.

Now add a second argument to the function `bar`. This requires you to edit `bar.c` and `bar.h`: go ahead and make these edits. However, it also requires you to edit `foo.c`, but let us for now 'forget' to do that. We will see how *Make* can help you find the resulting error.

Exercise. Call `make` to recompile your program. Did it recompile `foo.c`?

Expected outcome. Even through conceptually `foo.c` would need to be recompiled since it uses the `bar` function, *Make* did not do so because the makefile had no rule that forced it.

In the makefile, change the line

```
foo.o : foo.c
```

to

```
foo.o : foo.c bar.h
```

which adds `bar.h` as a prerequisite for `foo.o`. This means that, in this case where `foo.o` already exists, *Make* will check that `foo.o` is not older than any of its prerequisites. Since `bar.h` has been edited, it is younger than `foo.o`, so `foo.o` needs to be reconstructed.

Exercise. Confirm that the new makefile indeed causes `foo.o` to be recompiled if `bar.h` is changed. This compilation will now give an error, since you 'forgot' to edit the use of the `bar` function.

33.2.3 Fortran

Make the following files:

`foomain.F`

```
#include "papi_test.h"
extern int TESTS_QUIET;         /* Declared in test_utils.c */

#define PCHECK(e) if (e!=PAPI_OK) \
        {printf("Problem in papi call, line %d\n",__LINE__); \
         return 1;}
```

```
#define NEVENTS 2
/* we are assuming array storage in Fortran column mode */
#define INDEX(i,j,m,n) i+j*m

double *array;

void clear_right(int m,int n) {
  int i,j;
  for (j=0; j<n; j++)
    for (i=0; i<m; i++)
      array[INDEX(i,j,m,n)] = 0;
  return;
}

void clear_wrong(int m,int n) {
  int i,j;
  for (i=0; i<m; i++)
    for (j=0; j<n; j++)
      array[INDEX(i,j,m,n)] = 0;
  return;
}

void do_operation_right(int m,int n) {
  int i,j;
  for (j=0; j<n; j++)
    for (i=0; i<m; i++)
      array[INDEX(i,j,m,n)] = array[INDEX(i,j,m,n)]+1;
  return;
}

void do_operation_wrong(int m,int n) {
  int i,j;
  for (i=0; i<m; i++)
    for (j=0; j<n; j++)
      array[INDEX(i,j,m,n)] = array[INDEX(i,j,m,n)]+1;
  return;
}

#define COL 1
#define ROW 2
int main(int argc, char **argv)
{
   int events[NEVENTS] = {PAPI_TLB_DM,PAPI_TOT_CYC};
   long_long values[NEVENTS];
   int retval,order=COL;
   PAPI_event_info_t info, info1;
   const PAPI_hw_info_t *hwinfo = NULL;
   int event_code;
   const PAPI_substrate_info_t *s = NULL;

   tests_quiet(argc, argv);     /* Set TESTS_QUIET variable */
   if (argc==2 && !strcmp(argv[1],"row")) {
     printf("wrong way\n"); order=ROW;
   } else printf("right way\n");

   retval = PAPI_library_init(PAPI_VER_CURRENT);
   if (retval != PAPI_VER_CURRENT)
```

```
      test_fail(__FILE__, __LINE__, "PAPI_library_init", retval);

   {
     int i;
     for (i=0; i<NEVENTS; i++) {
       retval = PAPI_query_event(events[i]); PCHECK(retval);
     }
   }

#define M 1000
#define N 2000
   {
     int m,n;
     m = M;
     array = (double*) malloc(M*N*sizeof(double));
     for (n=10; n<N; n+=10) {
       if (order==COL)
 clear_right(m,n);
       else
 clear_wrong(m,n);
       retval = PAPI_start_counters(events,NEVENTS); PCHECK(retval);
       if (order==COL)
 do_operation_right(m,n);
       else
 do_operation_wrong(m,n);
       retval = PAPI_stop_counters(values,NEVENTS); PCHECK(retval);
       printf("m,n=%d,%d\n#elements:\t%d\n",m,n,m*n);
       printf("Tot cycles: %d\nTLB misses:\t%d\nmisses per column:\t%9.5f\n\n",
      values[1],values[0],values[0]/(1.*n));
     }
     free(array);
   }

   return 0;
}
```

foomod.F

```
#include "papi_test.h"
extern int TESTS_QUIET;         /* Declared in test_utils.c */

#define PCHECK(e) if (e!=PAPI_OK) \
        {printf("Problem in papi call, line %d\n",__LINE__); \
         return 1;}
#define NEVENTS 2
/* we are assuming array storage in Fortran column mode */
#define INDEX(i,j,m,n) i+j*m

double *array;

void clear_right(int m,int n) {
  int i,j;
  for (j=0; j<n; j++)
    for (i=0; i<m; i++)
      array[INDEX(i,j,m,n)] = 0;
  return;
}
```

```
void clear_wrong(int m,int n) {
  int i,j;
  for (i=0; i<m; i++)
    for (j=0; j<n; j++)
      array[INDEX(i,j,m,n)] = 0;
  return;
}

void do_operation_right(int m,int n) {
  int i,j;
  for (j=0; j<n; j++)
    for (i=0; i<m; i++)
      array[INDEX(i,j,m,n)] = array[INDEX(i,j,m,n)]+1;
  return;
}

void do_operation_wrong(int m,int n) {
  int i,j;
  for (i=0; i<m; i++)
    for (j=0; j<n; j++)
      array[INDEX(i,j,m,n)] = array[INDEX(i,j,m,n)]+1;
  return;
}

#define COL 1
#define ROW 2
int main(int argc, char **argv)
{
   int events[NEVENTS] = {PAPI_TLB_DM,PAPI_TOT_CYC};
   long_long values[NEVENTS];
   int retval,order=COL;
   PAPI_event_info_t info, info1;
   const PAPI_hw_info_t *hwinfo = NULL;
   int event_code;
   const PAPI_substrate_info_t *s = NULL;

   tests_quiet(argc, argv);     /* Set TESTS_QUIET variable */
   if (argc==2 && !strcmp(argv[1],"row")) {
     printf("wrong way\n"); order=ROW;
   } else printf("right way\n");

   retval = PAPI_library_init(PAPI_VER_CURRENT);
   if (retval != PAPI_VER_CURRENT)
      test_fail(__FILE__, __LINE__, "PAPI_library_init", retval);

   {
     int i;
     for (i=0; i<NEVENTS; i++) {
       retval = PAPI_query_event(events[i]); PCHECK(retval);
     }
   }

#define M 1000
#define N 2000
   {
     int m,n;
```

```
      m = M;
      array = (double*) malloc(M*N*sizeof(double));
      for (n=10; n<N; n+=10) {
        if (order==COL)
 clear_right(m,n);
        else
 clear_wrong(m,n);
        retval = PAPI_start_counters(events,NEVENTS); PCHECK(retval);
        if (order==COL)
 do_operation_right(m,n);
        else
 do_operation_wrong(m,n);
        retval = PAPI_stop_counters(values,NEVENTS); PCHECK(retval);
        printf("m,n=%d,%d\n#elements:\t%d\n",m,n,m*n);
        printf("Tot cycles: %d\nTLB misses:\t%d\nmisses per column:\t%9.5f\n\n",
       values[1],values[0],values[0]/(1.*n));
      }
      free(array);
    }

    return 0;
}
```

and a makefile:

Makefile

```
#include "papi_test.h"
extern int TESTS_QUIET;         /* Declared in test_utils.c */

#define PCHECK(e) if (e!=PAPI_OK) \
        {printf("Problem in papi call, line %d\n",__LINE__); \
         return 1;}
#define NEVENTS 2
/* we are assuming array storage in Fortran column mode */
#define INDEX(i,j,m,n) i+j*m

double *array;

void clear_right(int m,int n) {
  int i,j;
  for (j=0; j<n; j++)
    for (i=0; i<m; i++)
      array[INDEX(i,j,m,n)] = 0;
  return;
}

void clear_wrong(int m,int n) {
  int i,j;
  for (i=0; i<m; i++)
    for (j=0; j<n; j++)
      array[INDEX(i,j,m,n)] = 0;
  return;
}

void do_operation_right(int m,int n) {
  int i,j;
```

```
  for (j=0; j<n; j++)
    for (i=0; i<m; i++)
      array[INDEX(i,j,m,n)] = array[INDEX(i,j,m,n)]+1;
  return;
}

void do_operation_wrong(int m,int n) {
  int i,j;
  for (i=0; i<m; i++)
    for (j=0; j<n; j++)
      array[INDEX(i,j,m,n)] = array[INDEX(i,j,m,n)]+1;
  return;
}

#define COL 1
#define ROW 2
int main(int argc, char **argv)
{
   int events[NEVENTS] = {PAPI_TLB_DM,PAPI_TOT_CYC};
   long_long values[NEVENTS];
   int retval,order=COL;
   PAPI_event_info_t info, info1;
   const PAPI_hw_info_t *hwinfo = NULL;
   int event_code;
   const PAPI_substrate_info_t *s = NULL;

   tests_quiet(argc, argv);     /* Set TESTS_QUIET variable */
   if (argc==2 && !strcmp(argv[1],"row")) {
     printf("wrong way\n"); order=ROW;
   } else printf("right way\n");

   retval = PAPI_library_init(PAPI_VER_CURRENT);
   if (retval != PAPI_VER_CURRENT)
      test_fail(__FILE__, __LINE__, "PAPI_library_init", retval);

   {
     int i;
     for (i=0; i<NEVENTS; i++) {
       retval = PAPI_query_event(events[i]); PCHECK(retval);
     }
   }

#define M 1000
#define N 2000
   {
     int m,n;
     m = M;
     array = (double*) malloc(M*N*sizeof(double));
     for (n=10; n<N; n+=10) {
       if (order==COL)
 clear_right(m,n);
       else
 clear_wrong(m,n);
       retval = PAPI_start_counters(events,NEVENTS); PCHECK(retval);
       if (order==COL)
 do_operation_right(m,n);
       else
```

```
 do_operation_wrong(m,n);
       retval = PAPI_stop_counters(values,NEVENTS); PCHECK(retval);
       printf("m,n=%d,%d\n#elements:\t%d\n",m,n,m*n);
       printf("Tot cycles: %d\nTLB misses:\t%d\nmisses per column:\t%9.5f\n\n",
      values[1],values[0],values[0]/(1.*n));
     }
     free(array);
   }

   return 0;
}
```

If you call `make`, the first rule in the makefile is executed. Do this, and explain what happens.

Exercise. Call `make`.

> *Expected outcome.* The above rules are applied: `make` without arguments tries to build the first target, `foomain`. In order to build this, it needs the prerequisites `foomain.o` and `foomod.o`, which do not exist. However, there are rules for making them, which `make` recursively invokes. Hence you see two compilations, for `foomain.o` and `foomod.o`, and a link command for `fooprog`.
>
> *Caveats.* Typos in the makefile or in file names can cause various errors. Unfortunately, debugging a makefile is not simple. You will just have to understand the errors, and make the corrections.

Exercise. Do `make clean`, followed by `mv foomod.c boomod.c` and `make` again. Explain the error message. Restore the original file name.

> *Expected outcome. Make* will complain that there is no rule to make `foomod.c`. This error was caused when `foomod.c` was a prerequisite for `foomod.o`, and was found not to exist. *Make* then went looking for a rule to make it, and no rule for making `.F` files exists.

Now add an extra parameter to `func` in `foomod.F` and recompile.

Exercise. Call `make` to recompile your program. Did it recompile `foomain.F`?

> *Expected outcome.* Even through conceptually `foomain.F` would need to be recompiled, *Make* did not do so because the makefile had no rule that forced it.

Change the line

```
foomain.o : foomain.F
```

to

```
foomain.o : foomain.F foomod.o
```

which adds `foomod.o` as a prerequisite for `foomain.o`. This means that, in this case where `foomain.o` already exists, *Make* will check that `foomain.o` is not older than any of its prerequisites. Recursively, *Make* will then check if `foomode.o` needs to be updated, which is indeed the case. After recompiling `foomode.F`, `foomode.o` is younger than `foomain.o`, so `foomain.o` will be reconstructed.

Exercise. Confirm that the corrected makefile indeed causes `foomain.F` to be recompiled.

33.3 Some general remarks

33.3.1 Rule interpretation

Probably the best way to interpret a rule is:

- if any prerequisite has changed,
- then the target needs to be remade,
- and that is done by executing the commands of the rule;
- checking the prerequisite requires a recursive application of make:
  - if the prerequisite does not exist, find a rule to create it;
  - if the prerequisite already exists, check applicable rules to see if it needs to be remade.

33.3.2 Make invocation

If you call `make` without any arguments, the first rule in the makefile is evaluated. You can execute other rules by explicitly invoking them, for instance `make foo.o` to compile a single file.

33.3.3 About the make file

The make file needs to be called `makefile` or `Makefile`; it is not a good idea to have files with both names in the same directory. If you want *Make* to use a different file as make file, use the syntax `make -f My_Makefile`.

33.4 Variables and template rules

Purpose. In this section you will learn various work-saving mechanisms in *Make*, such as the use of variables and of template rules.

33.4.1 Makefile variables

It is convenient to introduce variables in your makefile. For instance, instead of spelling out the compiler explicitly every time, introduce a variable in the makefile:

```
CC = gcc
FC = gfortran
```

and use `${CC}` or `${FC}` on the compile lines:

```
foo.o : foo.c
        ${CC} -c foo.c
foomain.o : foomain.F
        ${FC} -c foomain.F
```

Exercise. Edit your makefile as indicated. First do `make clean`, then `make foo` (C) or `make fooprog` (Fortran).

Expected outcome. You should see the exact same compile and link lines as before.

Caveats. Unlike in the shell, where braces are optional, variable names in a makefile have to be in braces or parentheses. Experiment with what happens if you forget the braces around a variable name.

One advantage of using variables is that you can now change the compiler from the commandline:

```
make CC="icc -O2"
make FC="gfortran -g"
```

Exercise. Invoke *Make* as suggested (after `make clean`). Do you see the difference in your screen output?

Expected outcome. The compile lines now show the added compiler option `-O2` or `-g`.

Make also has *automatic variables*:

- `$@` The target. Use this in the link line for the main program.
- `$^` The list of prerequisites. Use this also in the link line for the program.
- `$<` The first prerequisite. Use this in the compile commands for the individual object files.
- `$*` In *template rules* (section 33.4.2) this matches the template part, the part corresponding to the `%`.

Using these variables, the rule for `fooprog` becomes

```
fooprog : foo.o bar.o
        ${CC} -o $@ $^
```

and a typical compile line becomes

```
foo.o : foo.c bar.h
        ${CC} -c $<
```

You can also declare a variable

```
THEPROGRAM = fooprog
```

and use this variable instead of the program name in your makefile. This makes it easier to change your mind about the name of the executable later.

Exercise. Edit your makefile to add this variable definition, and use it instead of the literal program name. Construct a commandline so that your makefile will build the executable `fooprog_v2`.

Expected outcome. You need to specify the `THEPROGRAM` variable on the commandline using the syntax `make VAR=value`.

Caveats. Make sure that there are no spaces around the equals sign in your commandline.

The full list of these automatic variables can be found at https://www.gnu.org/software/make/manual/html_node/Automatic-Variables.html.

33.4.2 Template rules

So far, you wrote a separate rule for each file that needed to be compiled. However, the rules for the various `.c` files are very similar:

- the rule header (`foo.o : foo.c`) states that a source file is a prerequisite for the object file with the same base name;
- and the instructions for compiling (`${CC} -c $<`) are even character-for-character the same, now that you are using *Make*'s built-in variables;
- the only rule with a difference is

```
foo.o : foo.c bar.h
        ${CC} -c $<
```

  where the object file depends on the source file and another file.

We can take the commonalities and summarize them in one *template rule*[1]:

```
%.o : %.c
        ${CC} -c $<
%.o : %.F
        ${FC} -c $<
```

This states that any object file depends on the C or Fortran file with the same base name. To regenerate the object file, invoke the C or Fortran compiler with the `-c` flag. These template rules can function as a replacement for the multiple specific targets in the makefiles above, except for the rule for `foo.o`.

The dependence of `foo.o` on `bar.h`, or `foomain.o` on `foomod.o`, can be handled by adding a rule

```
# C
foo.o : bar.h
# Fortran
foomain.o : foomod.o
```

with no further instructions. This rule states, 'if file `bar.h` or `foomod.o` changed, file `foo.o` or `foomain.o` needs updating' too. *Make* will then search the makefile for a different rule that states how this updating is done, and it will find the template rule.

Exercise. Change your makefile to incorporate these ideas, and test.

Exercise 33.1. Write a makefile for the following structure:

- There is one main file `libmain.cxx`, and two library files `libf.cxx` `libg.cxx`;
- There is a header file `libapi.h` that gives the prototypes for the functions in the library files;
- There is a header file `libimpl.h` that gives implementation details, only to be used in the library files.

This is illustrated in figure 33.1.

Here is how you can test it:

1. This mechanism is the first instance you'll see that only exists in GNU make, though in this particular case there is a similar mechanism in standard make. That will not be the case for the wildcard mechanism in the next section.

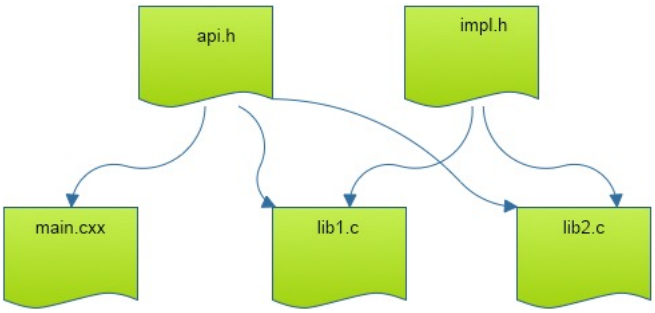


Figure 33.1: File structure with main program and two library files.

Source file `mainprog.cxx`:

```
#include <cstdio>
#include "api.h"

int main() {
    int n = f() + g();
    printf("%d\n",n);
    return 0;
}
```

Source file `libf.cxx`:

```
#include "impl.h"
#include "api.h"

int f() {
    struct impl_struct foo;
    return 1;
};
```

Source file `libg.cxx`:

```
#include "impl.h"
#include "api.h"

int g() {
    struct impl_struct foo;
    return 2;
};
```

Header file `api.h`:

```
int f();
int g();
```

Header file `impl.h`:

```
struct impl_struct {};
```

Figure 33.2: Source files for exercise 33.1.

Changing a source file only recompiles that files:

```
clang++ -c libf.cxx
clang++ -o main \
    libmain.o libf.o libg.o
```

Changing the implementation header only recompiles the library:

```
clang++ -c libf.cxx
clang++ -c libg.cxx
clang++ -o main libmain.o libf.o
    libg.o
```

Changing the `libapi.h` recompiles everything:

```
clang++ -c libmain.cxx
clang++ -c libf.cxx
clang++ -c libg.cxx
clang++ -o main libmain.o libf.o
    libg.o
```

For Fortran we don't have header files so we use *modules* everywhere; figure 33.3. If you know how to use *submodules*, a *Fortran2008* feature, you can make the next exercise as efficient as the C version.

Exercise 33.2. Write a makefile for the following structure:

- There is one main file `libmain.f90`, that uses a module `api.f90`;
- There are two low level modules `libf.f90` `libg.f90` that are used in `api.f90`.

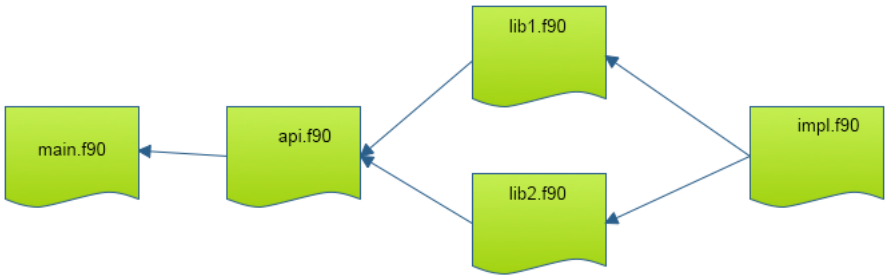


Figure 33.3: File structure with main program and two library files.

If you use modules, you'll likely be doing more compilation than needed. For the optimal solution, use submodules.

33.4.3 Wildcards

Your makefile now uses one general rule for compiling any source file. Often, your source files will be all the `.c` or `.F` files in your directory, so is there a way to state 'compile everything in this directory'? Indeed there is.

Add the following lines to your makefile, and use the variable `COBJECTS` or `FOBJECTS` wherever appropriate. The command *`wildcard`* gives the result of `ls`, and you can manipulate file names with *`patsubst`*.

```
# wildcard: find all files that match a pattern
CSOURCES := ${wildcard *.c}
# pattern substitution: replace one pattern string by another
COBJECTS := ${patsubst %.c,%.o,${SRC}}

FSOURCES := ${wildcard *.F}
FOBJECTS := ${patsubst %.F,%.o,${SRC}}
```

33.4.4 More functions

GNU make has more function that you can call inside the makefile. Some examples:

```
HOSTNAME := $(shell hostname -f)
SOURCES  := $(wildcard *.c)
OBJECTS  := $(patsubst %.c,%.o,${SOURCES})
RECURSIVE := $(foreach d,${DIRECTORIES},$(wildcard ${d}/*.c))
```

File name manipulation:

```
$(dir a/b/c.x)     # gives 'a/b'
$(dir c.x)         # gives './'
$(notdir a/b/c.x)  # gives 'c.x'
$(suffix a/b/c.x)  # gives '.x'
```

For the full list see https://www.gnu.org/software/make/manual/html_node/Functions.html.

```
// Makefile
SRC := ${wildcard src/*.c}

OBJ := ${patsubst
    src/%,obj/%,${patsubst
    %.c,%.o,${SRC}}}

PRE := ${addprefix /usr/lib,${SRC}
    othersrc moresrc}

BAK := ${addsuffix .bak,${SRC}}
```

```
SRC: src/f1.c src/f2.c

OBJ: obj/f1.o obj/f2.o

PRE: /usr/libsrc/f1.c
    /usr/libsrc/f2.c
    /usr/libothersrc /usr/libmoresrc

BAK: src/f1.c.bak src/f2.c.bak
```

33.4.5 Conditionals

There are various ways of making the behavior of a makefile dynamic. You can for instance put a shell conditional in an action line. However, this can make for a cluttered makefile; an easier way is to use makefile conditionals. There are two types of conditionals: tests on string equality, and tests on environment variables.

The first type looks like

```
ifeq "${HOME}" "/home/thisisme"
  # case where the executing user is me
else ifeq "${HOME}" "/home/buddyofmine"
  # case for other user
else
  # case where it's someone else
endif
```

and in the second case the test looks like

```
ifdef SOME_VARIABLE
```

The text in the true and false part can be most any part of a makefile. For instance, it is possible to let one of the action lines in a rule be conditionally included. However, most of the times you will use conditionals to make the definition of variables dependent on some condition.

Exercise. Let's say you want to use your makefile at home and at work. At work, your employer has a paid license to the Intel compiler `icc`, but at home you use the open source Gnu compiler `gcc`. Write a makefile that will work in both places, setting the appropriate value for `CC`.

33.5 Miscellania

33.5.1 Phony targets

The example makefile contained a target `clean`. This uses the *Make* mechanisms to accomplish some actions that are not related to file creation: calling `make clean` causes *Make* to reason 'there is no

file called `clean`, so the following instructions need to be performed'. However, this does not actually cause a file `clean` to spring into being, so calling `make clean` again will make the same instructions being executed.

To indicate that this rule does not actually make the target, you use the `.PHONY` keyword:

```
.PHONY : clean
```

Most of the time, the makefile will actually work fine without this declaration, but the main benefit of declaring a target to be phony is that the *Make* rule will still work, even if you have a file (or folder) named `clean`.

33.5.2 Directories

It's a common strategy to have a directory for temporary material such as object files. So you would have a rule

```
obj/%.o : %.c
    ${CC} -c $< -o $@
```

and to remove the temporaries:

```
clean ::
    rm -rf obj
```

This raises the question how the `obj` directory is created. You could do:

```
obj/%.o : %.c
    mkdir -p obj
    ${CC} -c $< -o $@
```

but a better solution is to use *order-only prerequisite*s exist.

```
obj :
    mkdir -p obj
obj/%.o : %.c | obj
    ${CC} -c $< -o $@
```

This only tests for the existence of the object directory, but not its timestamp.

33.5.3 Using the target as prerequisite

Suppose you have two different targets that are treated largely the same. You would want to write:

```
PROGS = myfoo other
${PROGS} : $@.o # this is wrong!!
        ${CC} -o $@ $@.o ${list of libraries goes here}
```

and saying `make myfoo` would cause

```
cc -c myfoo.c
cc -o myfoo myfoo.o ${list of libraries}
```

and likewise for `make other`. What goes wrong here is the use of `$@.o` as prerequisite. In Gnu Make, you can repair this as follows[2]:

```
.SECONDEXPANSION:
${PROGS} : $$@.o
        ${CC} -o $@ $@.o ${list of libraries goes here}
```

Exercise. Write a second main program `foosecond.c` or `foosecond.F`, and change your makefile so that the calls `make foo` and `make foosecond` both use the same rule.

33.5.4 Predefined variables and rules

Calling `make -p yourtarget` causes make to print out all its actions, as well as the values of all variables and rules, both in your makefile and ones that are predefined. If you do this in a directory where there is no makefile, you'll see that make actually already knows how to compile `.c` or `.F` files. Find this rule and find the definition of the variables in it.

You see that you can customize make by setting such variables as `CFLAGS` or `FFLAGS`. Confirm this with some experimentation. If you want to make a second makefile for the same sources, you can call `make -f othermakefile` to use this instead of the default `Makefile`.

Note, by the way, that both `makefile` and `Makefile` are legitimate names for the default makefile. It is not a good idea to have both `makefile` and `Makefile` in your directory.

33.6 Shell scripting in a Makefile

Purpose. In this section you will see an example of a longer shell script appearing in a makefile rule.

In the makefiles you have seen so far, the command part was a single line. You can actually have as many lines there as you want. For example, let us make a rule for making backups of the program you are building.

Add a `backup` rule to your makefile. The first thing it needs to do is make a backup directory:

```
.PHONY : backup
backup :
        if [ ! -d backup ] ; then
          mkdir backup
        fi
```

Did you type this? Unfortunately it does not work: every line in the command part of a makefile rule gets executed as a single program. Therefore, you need to write the whole command on one line:

```
backup :
        if [ ! -d backup ] ; then mkdir backup ; fi
```

2. Technical explanation: Make will now look at lines twice: the first time `$$` gets converted to a single `$`, and in the second pass `$@` becomes the name of the target.

or if the line gets too long:

```
backup :
        if [ ! -d backup ] ; then \
          mkdir backup ; \
        fi
```

(Writing a long command on a single is only possible in the *bash* shell, not in the *csh* shell. This is one reason for not using the latter.)

Next we do the actual copy:

```
backup :
        if [ ! -d backup ] ; then mkdir backup ; fi
        cp myprog backup/myprog
```

But this backup scheme only saves one version. Let us make a version that has the date in the name of the saved program.

The Unix `date` command can customize its output by accepting a format string. Type the following: `date` This can be used in the makefile.

Exercise. Edit the `cp` command line so that the name of the backup file includes the current date.

> *Expected outcome.* Hint: you need the backquote. Consult the Unix tutorial, section **??**, if you do not remember what backquotes do.

If you are defining shell variables in the command section of a makefile rule, you need to be aware of the following. Extend your `backup` rule with a loop to copy the object files:

```
#### This Script Has An ERROR!
backup :
        if [ ! -d backup ] ; then mkdir backup ; fi
        cp myprog backup/myprog
        for f in ${OBJS} ; do \
          cp $f backup ; \
        done
```

(This is not the best way to copy, but we use it for the purpose of demonstration.) This leads to an error message, caused by the fact that *Make* interprets `$f` as an environment variable of the outer process. What works is:

```
backup :
        if [ ! -d backup ] ; then mkdir backup ; fi
        cp myprog backup/myprog
        for f in ${OBJS} ; do \
          cp $$f backup ; \
        done
```

(In this case *Make* replaces the double dollar by a single one when it scans the commandline. During the execution of the commandline, `$f` then expands to the proper filename.)

33.7 Practical tips for using Make

Here are a couple of practical tips.

- *Debugging* a makefile is often frustratingly hard. Just about the only tool is the `-p` option, which prints out all the rules that Make is using, based on the current makefile.
- You will often find yourself first typing a make command, and then invoking the program. Most Unix shells allow you to use commands from the *shell command history* by using the up arrow key. Still, this may get tiresome, so you may be tempted to write

  ```
  make myprogram ; ./myprogram -options
  ```

  and keep repeating this. There is a danger in this: if the make fails, for instance because of compilation problems, your program will still be executed. Instead, write

  ```
  make myprogram && ./myprogram -options
  ```

  which executes the program conditional upon make concluding successfully.

33.7.1 What does this makefile do?

Above you learned that issuing the `make` command will automatically execute the first rule in the makefile. This is convenient in one sense[3], and inconvenient in another: the only way to find out what possible actions a makefile allows is to read the makefile itself, or the – usually insufficient – documentation.

A better idea is to start the makefile with a target

```
info :
        @echo "The following are possible:"
        @echo "  make"
        @echo "  make clean"
```

Now `make` without explicit targets informs you of the capabilities of the makefile.

If your makefile gets longer, you might want to document each section like this. This runs into a problem: you can not have two rules with the same target, `info` in this case. However, if you use a double colon it *is* possible. Your makefile would have the following structure:

```
info ::
        @echo "The following target are available:"
        @echo "  make install"
install :
        # ..... instructions for installing
info ::
       @echo "  make clean"
clean :
       # ..... instructions for cleaning
```

33.8 A Makefile for LaTeX

The *Make* utility is typically used for compiling programs, but other uses are possible too. In this section, we will discuss a makefile for LaTeX documents.

We start with a very basic makefile:

3. There is a convention among software developers that a package can be installed by the sequence `./configure ; make ; make install`, meaning: Configure the build process for this computer, Do the actual build, Copy files to some system directory such as `/usr/bin`.

```
info :
        @echo "Usage: make foo"
        @echo "where foo.tex is a LaTeX input file"

%.pdf : %.tex
        pdflatex $<
```

The command `make myfile.pdf` will invoke `pdflatex myfile.tex`, if needed, once. Next we repeat invoking `pdflatex` until the log file no longer reports that further runs are needed:

```
%.pdf : %.tex
        pdflatex $<
        while [ `cat ${basename $@}.log | grep "Rerun to get" \
            | wc -l` -gt 0 ] ; do \
          pdflatex $< ; \
        done
```

We use the `${basename fn}` macro to extract the base name without extension from the target name.

In case the document has a bibliography or index, we run `bibtex` and `makeindex`.

```
%.pdf : %.tex
        pdflatex ${basename $@}
        -bibtex ${basename $@}
        -makeindex ${basename $@}
        while [ `cat ${basename $@}.log | grep "Rerun to get" \
                 | wc -l` -gt 0 ] ; do \
          pdflatex ${basename $@} ; \
        done
```

The minus sign at the start of the line means that *Make* should not exit if these commands fail.

Finally, we would like to use *Make*'s facility for taking dependencies into account. We could write a makefile that has the usual rules

```
mainfile.pdf : mainfile.tex includefile.tex
```

but we can also discover the include files explicitly. The following makefile is invoked with

```
make pdf TEXTFILE=mainfile
```

The `pdf` rule then uses some shell scripting to discover the include files (but not recursively), and it calls *Make* again, invoking another rule, and passing the dependencies explicitly.

```
pdf :
        export includes=`grep "^.input " ${TEXFILE}.tex \
            | awk '{v=v FS $$2".tex"} END {print v}'` ; \
        ${MAKE} ${TEXFILE}.pdf INCLUDES="$$includes"

%.pdf : %.tex ${INCLUDES}
        pdflatex $< ; \
        while [ `cat ${basename $@}.log \
                | grep "Rerun to get" | wc -l` -gt 0 ] ; do \
          pdflatex $< ; \
        done
```

This shell scripting can also be done outside the makefile, generating the makefile dynamically.

Chapter 34

The CMake eco-system

Chapter 35

External libraries

If you have a C++ compiler, you can write as much software as your want, by yourself. However, some things that you may need for your work have already been written by someone else. How can you use their software?

35.1 What are software libraries?

In this chapter you will learn about the use of *software libraries*: software that is written not as a standalone package, but in such a way that you can access its functionality in your own program.

Software libraries can be enormous, as is the case for scientific libraries, which are often multi-person multi-year projects. On the other hand, many of them are fairly simple utilities written by a single programmer. In the latter case you may have to worry about future support of that software.

35.1.1 Using an external library

Using a software library typically means that

- your program has a line

```
#include "fancylib.h"
```

- and you compile and link as:

```
icpc -c yourprogram.cpp -I/usr/include/fancylib
icpc -o yourprogram yourprogram.o -L/usr/lib/fancylib -lfancy
```

You will see specific examples below.

If you are now worried about having to do a lot of typing every time you compile,

- if you use an Integrated Development Environment (IDE), you can typically add the library in the options, once and for all; or
- you can use *Make* for building your program. See the tutorial.

35.1.2 Obtaining and installing an external library

Sometimes a software library is available through a *package manager*, but we are going to do it the old-fashioned way: downloading and installing it ourselves.

A popular location for finding downloadable software is `github.com`. You can then choose whether to

- clone the repository, or
- download everything in one file, typically with `.tgz` or `.tar.gz` extension; in that case you need to unpack it
  ```
  tar fxz fancylib.tgz
  ```
  This usually gives you a directory with a name such as
  ```
  fancylib-1.0.0
  ```
  containing the source and documentation of the library, but not any binaries or machine-specific files.

Either way, from here on we assume that you have a directory containing the downloaded package.

There are two main types of installation:

- based on *GNU autotools*, which you recognize by the presence of a *configure* program;
  ```
  cmake ## lots of options
  make
  make install
  ```
  or
- based on *Cmake*, which you recognize by the presence of `CMakeLists.txt` file:
  ```
  configure ## lots of options
  make
  make install
  ```

35.1.2.1 Cmake installation

The easiest way to install a package using `cmake` is to create a build directory, next to the source directory. The `cmake` command is issued from this directory, and it references the source directory:

```
mkdir build
cd build
cmake ../fancylib-1.0.0
make
make install
```

Some people put the build directory inside the source directory, but that is bad practice.

Apart from specifying the source location, you can give more options to `cmake`. The most common are

- specifying an install location, for instance because you don't have *superuser* privileges on that machine; or

- specifying the compiler, because Cmake will be default use the *gcc* compilers, but you may want the Intel compiler.

```
CC=icc CXX=icpc \
cmake \
    -D CMAKE_INSTALL_PREFIX:PATH=${HOME}/mylibs/fancy \
    ../fancylib-1.0.0
```

35.2 Options processing: cxxopts

Suppose you have a program that does something with a large array, and you want to be able to change your mind about the array size. You could

- You could recompile your program every time.
- You could let your program parse *argv*, and hope you remember precisely how your commandline options are to be interpreted.
- You could use the *cxxopts* library. This is what we will be exploring now.

35.2.1 Traditional commandline parsing

Commandline options are available to the program through the (optional) *argv* and *argc* options of the *main* program. The former is an array of strings, with the second the length:

```
int main( int argc,char **argv ) { /* program */ };
```

For simple cases it would be feasible to parse such options yourself:

Code:

```
cout << "Program name: " << argv[0] <<
     '\n';
for (int iarg=1; iarg<argc; ++iarg)
  cout << "arg: " << iarg
       << argv[iarg] << " => "
       << atoi( argv[iarg] ) << '\n';
```

Output
[args] argcv:

```
./argcv 5 12
Program name: ./argcv
arg 1: 5 => 5
arg 2: 12 => 12
./argcv abc 3.14 foo
Program name: ./argcv
arg 1: abc => 0
arg 2: 3.14 => 3
arg 3: foo => 0
```

but this 1. gets tedious quickly 2. is difficult to make robust. Therefore, we will now see a library that makes such options handling relatively easy.

35.2.2 The cxxopts library

The *cxxopts* 'commandline argument parser' can be found at https://github.com/jarro2783/cxxopts. After a *Cmake* installation, it is a 'header-only' library.

- Include the header

```
#include "cxxopts.hpp"
```

which requires a compile option:

```
-I/path/to//cxxopts/installdir/include
```

- Declare an options object:

```
cxxopts::Options options("programname", "Program description");
```

- Add options:

```
// define `-n 567' option:
options.add_options()
  ("n,ntimes","number of times",
   cxxopts::value<int>()->default_value("37")
   )
  ;
  /* ... */
// read out `-n' option and use:
auto number_of_times = result["ntimes"].as<int>();
cout << "Using number of times: " << number_of_times << '\n';
```

- Add array options

```
//define `-a 1,2,5,7' option:
options.add_options()
  ("a,array","array of values",
   cxxopts::value< vector<int> >()->default_value("1,2,3")
   )
  ;
  /* ... */
auto array = result["array"].as<vector<int>>();
cout << "Array: " ;
for ( auto a : array ) cout << a << ", ";
cout << '\n';
```

- Add positional arguments:

```
// define `positional argument' option:
options.add_options()
  ("keyword","whatever keyword",
   cxxopts::value<string>())
  ;
options.parse_positional({"keyword"});
  /* ... */
// read out keyword option and use:
auto keyword = result["keyword"].as<string>();
cout << "Found keyword: " << keyword << '\n';
```

- Parse the options:

```
auto result = options.parse(argc, argv);
```

- Get help flag first:

```
options.add_options()
  ("h,help","usage information")
  ;
  /* ... */
auto result = options.parse(argc, argv);
```

```
if (result.count("help")>0) {
  cout << options.help() << '\n';
  return 0;
}
```

- Get result values for both options and arguments:

```
auto number_of_times = result["ntimes"].as<int>();
cout << "Using number of times: " << number_of_times << '\n';

auto keyword = result["keyword"].as<string>();
cout << "Found keyword: " << keyword << '\n';
```

Options can be specified the usual ways:

```
myprogram -n 10
myprogram --nsize 100
myprogram --nsize=1000
myprogram --array 12,13,14,15
```

Exercise 35.1. Incorporate this package into primality testing: exercise 17.25.

Options parsing can throw a `cxxopts::exceptions::option_has_no_value` exception.

35.2.3 Cmake integration

The `cxxopts` package can be discovered in *CMake* through `pkgconfig`:

```
find_package( PkgConfig REQUIRED )
pkg_check_modules( CXXOPTS REQUIRED cxxopts )
target_include_directories( ${PROJECT_NAME} PUBLIC ${CXXOPTS_INCLUDE_DIRS} )
```

35.3 Catch2 unit testing

Test a simple function

```
int five() { return 5; }
```

Successful test:

Code:

```
1 TEST_CASE( "needs to be 5","[1]" ) {
2     REQUIRE( five()==5 );
3 }
```

Output

```
[catch] require:

Filters: [1]
===============================================================
All tests passed (1
    assertion in 1 test case)
```

Unsuccessful test:

Code:

```
1 TEST_CASE( "not six","[2]" ) {
2     REQUIRE( five()==6 );
3 }
```

Output
[catch] requirerr:

```
require.cxx:30: FAILED:
  REQUIRE( five()==6 )
with expansion:
  5 == 6

===============================================================
test cases: 1 | 1 failed
assertions: 1 | 1 failed
```

Function that throws:

```
void even( int e ) {
    if (e%2==1) throw(1);
    cout << "Even number: "
         << e << '\n';
}
```

Test that it throws or not:

Code:

```
1 TEST_CASE( "even fun","[3]" ) {
2     REQUIRE_NOTHROW( even(2) );
3     REQUIRE_THROWS( even(3) );
4 }
```

Output
[catch] requireven:

```
Filters: [3]
Even number: 2
===============================================================
All tests passed (2
    assertions in 1 test case)
```

Run the same test for a set of numbers:

Code:

```
1 TEST_CASE( "even set","[4]" ) {
2     int e = GENERATE( 2,4,6,8 );
3     REQUIRE_NOTHROW( even(e) );
4 }
```

Output
[catch] requirgen:

```
Filters: [4]
Even number: 2
Even number: 4
Even number: 6
Even number: 8
===============================================================
All tests passed (4
    assertions in 1 test case)
```

How is this different from using a loop? Using `GENERATE` runs each value as a separate program.

Variants:

```
int i = GENERATE( range(1,100) );
int i = GENERATE_COPY( range(1,n) );
```

Chapter 36

Unit testing and Test-Driven Development

In an ideal world, you would prove your program correct, but in practice that is not always feasible, or at least: not done. Most of the time programmers establish the correctness of their code by testing it.

Yes, there is a quote by *Edsger Dijkstra* that goes:

> Today a usual technique is to make a program and then to test it. But: program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence. (cue laughter)

but that doesn't mean that you can't at least gain some confidence in your code by testing it.

36.1 Types of tests

Testing code is an art, even more than writing the code to begin with. That doesn't mean you can't be systematic about it. First of all, we distinguish between some basic types of test:

- *Unit tests* that test a small part of a program by itself;
- *System tests* test the correct behavior of the whole software system; and
- *Regression tests* establish that the behavior of a program has not changed by adding or changing aspects of it.

In this section we will talk about unit testing.

A program that is written in a sufficiently modular way allows for its components to be tested without having to wait for an all-or-nothing test of the whole program. Thus, testing and program design are aligned in their interests. In fact, writing a program with the thought in mind that it needs to be testable can lead to cleaner, more modular code.

In an extreme form of this you would write your code by Test-Driven Development (TDD), where code development and testing go hand-in-hand. The basic principles can be stated as follows:

- Both the whole code and its parts should always be testable.
- When extending the code, make only the smallest change that allows for testing.
- With every change, test before and after.
- Assure correctness before adding new features.

In a strict interpretation, you would even for each part of the program first write the test that it would satisfy, and then the actual code.

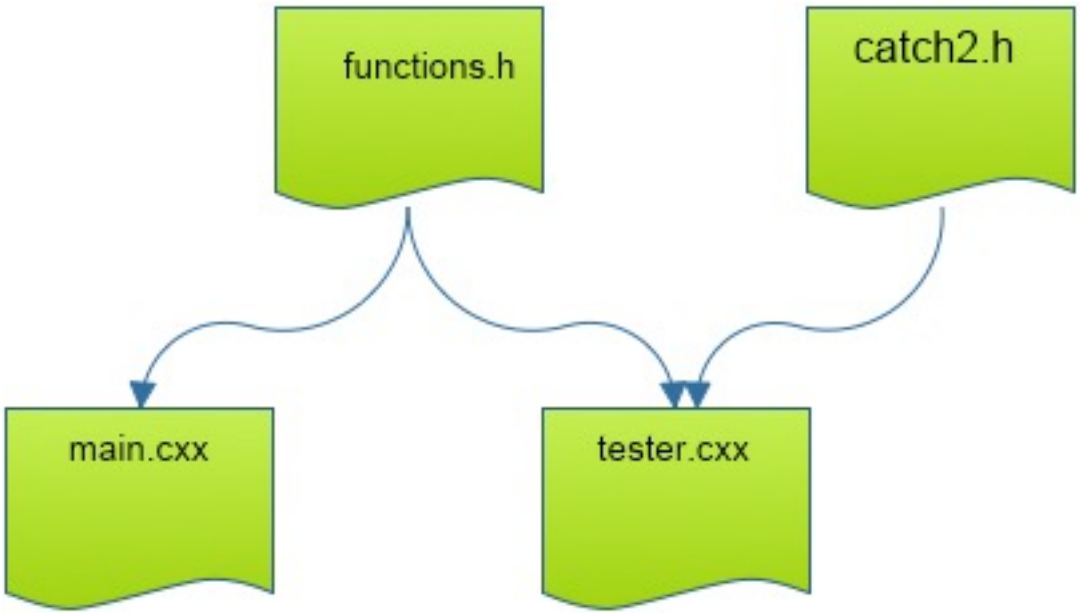


Figure 36.1: File structure for unit tests

36.2 Unit testing frameworks

There are several 'frameworks' that help you with unit testing. In the remainder of this chapter we will use *Catch2*, which is one of the most used ones in C++.

You can find the code at https://github.com/catchorg.

36.2.1 File structure

Let's assume you have a file structure with

- a very short main program, and
- a library file that has all functions used by the main.

In order to test the functions, you supply another main, which contains only unit tests; This is illustrated in figure 36.1.

In fact, with Catch2 your main file doesn't actually have a `main` program: that is supplied by the framework. In the tester main file you only put the test cases.

The framework supplies its own main:

```
#define CATCH_CONFIG_MAIN
#include "catch2/catch_all.hpp"

#include "library_functions.h"
/*
   here follow the unit tests
*/
```

One important question is what header file to include. You can do

```
#include "catch.hpp"
```

which is the 'header only' mode, but that makes compilation very slow. Therefore, we will assume you have installed Catch through *Cmake*, and you include

```
#include "catch2/catch_all.hpp"
```

Note: as of September 2021 this requires the development version of the repository, not any `2.x` release.

36.2.2 Compilation

The setup suggested above requires you to add compile and link flags to your setup. This is system-dependent.

One-line solution:

```
icpc -o tester test_main.cpp \
    -I${TACC_CATCH2_INC} -L${TACC_CATCH2_LIB} \
    -lCatch2Main -lCatch2
```

Variables for a Makefile:

```
INCLUDES = -I${TACC_CATCH2_INC}
EXTRALIBS = -L${TACC_CATCH2_LIB} -lCatch2Main -lCatch2
```

36.2.3 Test cases

A test case is a short program that is run as an independent main. In the setup suggested above, you put all your unit tests in the tester main program, that is, the file that has the

```
#define CATCH_CONFIG_MAIN
#include "catch2/catch_all.hpp"
```

magic lines.

Each test case needs to have a unique name, which is printed when a test fails. You can optionally add keys to the test case that allow you to select tests from the commandline.

```
TEST_CASE( "name of this test" ) {
  // stuf
}
TEST_CASE( "name of this test","[key1][key2]" ) {
  // stuf
}
```

The body of the test case is essentially a main program, where some statements are encapsulated in test macros. The most common macro is `REQUIRE`, which is used to demand correctness of some condition.

Tests go in `tester.cpp`:

```
TEST_CASE( "test that f always returns positive" ) {
  for (int n=0; n<1000; n++)
    REQUIRE( f(n)>0 );
}
```

- `TEST_CASE` acts like independent main program.
  can have multiple cases in a tester file
- `REQUIRE` is like `assert` but more sophisticated

Exercise 36.1.

1. Write a function

   ```
   double f(int n) { /* .... */ }
   ```

   that takes on positive values only.
2. Write a unit test that tests the function for a number of values.

You can base this off the file `tdd.cxx` in the repository

Boolean:

```
REQUIRE( some_test(some_input) );
REQUIRE( not some_test(other_input) );
```

Integer:

```
REQUIRE( integer_function(1)==3 );
REQUIRE( integer_function(1)!=0 );
```

Beware floating point:

```
REQUIRE( real_function(1.5)==Catch::Approx(3.0) );
REQUIRE( real_function(1)!=Catch::Approx(1.0) );
```

In general exact tests don't work.

For failing tests, the framework will give the name of the test, the line number, and the values that were tested.

Run the tester:

```
-------------------------------
test the increment function
-------------------------------
test.cpp:25
...............................

test.cpp:29: FAILED:
  REQUIRE( increment_positive_only(i)==i+1 )
with expansion:
  1 == 2
```

```
      ================================
      test cases: 1 | 1 failed
      assertions: 1 | 1 failed
```

In the above case, the error message printed out the offending value of `f(n)`, not the value of `n` for which it occurs. To determine this, insert `INFO` specifications, which only get print out if a test fails.

`INFO`: print out information at a failing test

```
TEST_CASE( "test that f always returns positive" ) {
  for (int n=0; n<1000; n++)
    INFO( "function fails for " << n );
    REQUIRE( f(n)>0 );
}
```

If your code throws exceptions (section 13.2.2) you can test for these.

Suppose function `g(n)`

- succeeds for input $n > 0$
- fails for input $n \leq 0$:
  throws exception

```
TEST_CASE( "test that g only works for positive" ) {
  for (int n=-100; n<+100; n++)
    if (n<=0)
      REQUIRE_THROWS( g(n) );
    else
      REQUIRE_NOTHROW( g(n) );
}
```

A common occurrence in unit testing is to have multiple tests with a common setup or tear down, to use terms that you sometimes come across in unit testing. Catch2 supports this: you can make 'sections' for part in between setup and tear down.

Use `SECTION` if tests have intro/outtro in common:

```
TEST_CASE( "commonalities" ) {
  // common setup:
  double x,y,z;
  REQUIRE_NOTHROW( y = f(x) );
  // two independent tests:
  SECTION( "g function" ) {
    REQUIRE_NOTHROW( z = g(y) );
  }
  SECTION( "h function" ) {
    REQUIRE_NOTHROW( z = h(y) );
  }
  // common followup
  REQUIRE( z>x );
}
```

(sometimes called setup/teardown)

36.3 Example: zero-finding by bisection

Development of the zero-finding algorithm by bisection can be found in section 19.1.

36.4 An example: quadratic equation roots

We revisit exercise **??**, which used `std::variant` to return 0,1,2 roots of a quadratic equation. Here we use TDD to arrive at the code.

Throughout, we represent the polynomial

$$ax^2 + bx + c$$

as

```
using quadratic = tuple<double,double,double>;
```

When needed, you can unpack this again with

```
auto [a,b,c] = coefficients;
```

(Can you think of an `assert` statement here that might be useful?)

Exercise 36.2. Write a function

```
double discriminant( quadratic coefficients );
```

that computes $b^2 - 4ac$, and test:

```
TEST_CASE( "discriminant" ) {
  REQUIRE( discriminant( make_tuple(0., 2.5, 0.) ) ==Catch::Approx(6.25) );
  REQUIRE( discriminant( make_tuple(1., 0., 1.5 ) ) ==Catch::Approx(-6.) );
  REQUIRE( discriminant( make_tuple(.1, .1, .1*.5 ) ) ==Catch::Approx(-.01) );
}
```

It may be illustrative to see what happens if you leave out the approximate equality test:

```
REQUIRE( discriminant( make_tuple(.1, .1, .1*.5 ) ) == -.01 );
```

With this function it becomes easy to detect the case of no roots: the discriminant $D < 0$. Next we need to have the criterium for single or double roots: we have a single root if $D = 0$.

Exercise 36.3. Write a function

```
bool discriminant_zero( quadratic coefficients );
```

that passes the test

```
quadratic coefficients = make_tuple(a,b,c);
d = discriminant( coefficients );
z = discriminant_zero( coefficients );
INFO( a << "," << b << "," << c << " d=" << d );
REQUIRE( z );
```

Using for instance the values:

```
a = 2; b = 4; c = 2;
a = 2; b = sqrt(40); c = 5; // !!!
a = 3; b = 0; c = 0.;
```

This exercise is the first one where we run into numerical subtleties. The second set of test values has the discriminant zero in exact arithmetic, but nonzero in computer arithmetic. Therefore, we need to test whether it is small enough, compared to b.

Exercise 36.4. Be sure to also test the case where `discriminant_zero` returns false.

Now that we've detected a single root, we need the function that computes it. There are no subtleties in this one.

Exercise 36.5. Write the function `simple_root` that returns the single root. For confirmation, test

```
auto r = simple_root(coefficients);
REQUIRE( evaluate(coefficients,r)==Catch::Approx(0.).margin(1.e-14) );
```

The remaining case of two distinct roots is arrived at by elimination, and the only thing to do is write the function that returns them.

Exercise 36.6. Write a function that returns the two roots as a indexcstdpair:

```
pair<double,double> double_root( quadratic coefficients );
```

Test:

```
quadratic coefficients = make_tuple(a,b,c);
auto [r1,r2] = double_root(coefficients);
auto
  e1 = evaluate(coefficients,r1),
  e2 = evaluate(coefficients,r2);
REQUIRE( evaluate(coefficients,r1)==Catch::Approx(0.).margin(1.e-14) );
REQUIRE( evaluate(coefficients,r2)==Catch::Approx(0.).margin(1.e-14) );
```

The final bit of code is the function that tests for how many roots there are, and returns them as a `std::variant`.

Exercise 36.7. Write a function

```
variant< bool,double, pair<double,double> >
    compute_roots( quadratic coefficients);
```

Test:

```
TEST_CASE( "full test" ) {
  double a,b,c; int index;
  SECTION( "no root" ) {
    a=2.0; b=1.5; c=2.5;
    index = 0;
  }
  SECTION( "single root" ) {
    a=1.0; b=4.0; c=4.0;
    index = 1;
  }
  SECTION( "double root" ) {
    a=2.2; b=5.1; c=2.5;
    index = 2;
  }
  quadratic coefficients =
    make_tuple(a,b,c);
  auto result =
    compute_roots(coefficients);
  REQUIRE( result.index()==index );
}
```

36.5 Eight queens example

See 20.3.

Chapter 37

Debugging with gdb

37.1 A simple example

The following program does not have any bugs; we use it to show some of the basics of gdb.

```
void say(int n) {
  cout << "hello world " << n << '\n';
}

int main() {

  for (int i=0; i<10; ++i) {
    int ii;
    ii = i*i;
    ++ii;
    say(ii);
  }

  return 0;
}
```

37.1.1 Invoking the debugger

After you compile your program, instead of running it the normal way, you invoke *gdb*:

```
gdb myprogram
```

That puts you in an environment, recognizable by the `(gdb)` prompt:

```
GNU gdb (GDB) Red Hat Enterprise Linux 7.6.1-115.el7
[stuff]
(gdb)
```

where you can do a controlled run of your program with the *run* command:

```
(gdb) run
Starting program: /home/eijkhout/gdb/hello
hello world 1
hello world 2
hello world 5
```

```
hello world 10
hello world 17
hello world 26
hello world 37
hello world 50
hello world 65
hello world 82
[Inferior 1 (process 30981) exited normally]
```

37.2 Example: integer overflow

The following program shows *integer overflow*. (We are using `short` to force this to happen soon.)

Code:

```
1 void say(short n) {
2   cout << "hello world " << n << '\n';
3 }
4
5 int main() {
6
7   for (short i=0; ; i+=20) {
8     short ii;
9     ii = i*i;
10    ++i;
11    say(ii);
12  }
13
14  return 0;
15 }
```

Output
[gdb] hello:

```
hello world 1
hello world 401
hello world 1601
hello world 3601
hello world 6401
hello world 10001
hello world 14401
hello world 19601
hello world 25601
hello world 32401
hello world -25535
hello world -17135
hello world -7935
hello world 2065
hello world 12865
hello world 24465
hello world -28671
hello world -15471
hello world -1471
hello world 13329
```

37.3 More gdb

37.3.1 Run with commandline arguments

This program is self-contained, but if you had a program that takes *commandline arguments*:

```
./myprogram 25
```

you can supply those in gdb:

```
(gdb) run 25
```

37.3.2 Source listing and proper compilation

Inside gdb, you can get a source listing with the *list* command.

Let's try our program again:

```
[] icpc -o hello hello.cpp
[] gdb hello
GNU gdb (GDB) Red Hat Enterprise Linux 7.6.1-115.el7
Copyright (C) 2013 Free Software Foundation, Inc.
Reading symbols from /home/eijkhout/gdb/hello...
    (no debugging symbols found)...
done.
(gdb) list
No symbol table is loaded.  Use the "file" command.
```

See the repeated reference to 'symbols'? You need to supply the `-g` compiler option for the *symbol table* to be included in the binary:

```
[] icpc -g -o hello hello.cpp
[] gdb hello
GNU gdb (GDB) Red Hat Enterprise Linux 7.6.1-115.el7
[stuff]
Reading symbols from /home/eijkhout/gdb/hello...done.
(gdb) list
13      using std::cout;
14      using std::endl;
[et cetera]
```

(If you hit return now, the list command is repeated and you get the next block of lines. Doing `list -` gives you the block above where you currently are.)

37.3.3 Stepping through the source

Let's now make a more controlled run of the program. In the source, we see that line 22 is the first executable one:

```
20      int main() {
21
22        for (int i=0; i<10; i++) {
23          int ii;
24          ii = i*i;
...
```

We introduce a *breakpoint* with the *break* command:

```
(gdb) break 22
Breakpoint 1 at 0x400a03: file hello.cpp, line 22.
```

(If your program is spread over multiple files, you can specify the file name: `break otherfile.cpp:34`.)

Now if we run the program, it will stop at that line:

```
(gdb) run
Starting program: /home/eijkhout/gdb/hello

Breakpoint 1, main () at hello.cpp:22
22        for (int i=0; i<10; i++) {
```

To be precise: the program is stopped in the state before it executes this line.

We can now use *cont* (for 'continue') to let the program run on. Since there are no further breakpoints, the program will run to completion. This is not terribly useful, so let us change our minds about the location of the breakpoint: it would be more useful if the execution stopped at the start of every iteration.

Recall that the breakpoint had a number of 1, so we use *delete* to remove it, and we set a breakpoint inside the loop body instead, and continue until we hit it.

```
(gdb) delete 1
(gdb) break 23
Breakpoint 2 at 0x400a29: file hello.cpp, line 23.
(gdb) cont
Continuing.
Breakpoint 2, main () at hello.cpp:24
24          ii = i*i;
```

(Note that line 23 is not executable, so execution stops on the first line after that.)

Now if we continue, the program runs until the next break point:

```
(gdb) cont
Continuing.
hello world 5

Breakpoint 1, main () at hello.cpp:24
24          ii = i*i;
```

To get to the next statement, we use *next*:

```
(gdb) next
25          ii++;
(gdb)
```

Hitting return re-executes the previous command, so we go to the next line:

```
(gdb)
26          say(ii);
(gdb)
hello world 2

Breakpoint 1, main () at hello.cpp:24
24          ii = i*i;
```

You observe that the function call

1. is executed, as is clear from the `hello world 1` output, but
2. is not displayed in detail in the debugger.

The conclusion is that `next` goes to the next executable statement in the current subprogram, not into functions and such that get called from it.

If you want to go into the function `say`, you need to use *`step`*:

```
(gdb) next
25          ii++;
(gdb) next
26          say(ii);
(gdb) step
say (n=10) at hello.cpp:17
17        cout << "hello world " << n << endl;
```

The debugger reports the function name, and the names and values of the arguments. Another 'step' executes the current line and brings us to the end of the function, and the next 'step' puts us back in the main program:

```
(gdb)
hello world 10
18      }
(gdb)
main () at hello.cpp:24
24          ii = i*i;
```

37.3.4 Inspecting values

When execution is stopped at a line (remember, that means right before it is executed!) you can inspect any values in that subprogram:

```
24          ii = i*i;
(gdb) print i
$1 = 4
```

You can even let expressions be evaluated with local variables:

```
(gdb) print 2*i
$2 = 8
```

You can combine this looking at values with breakpoints. Say you want to know when the variable `ii` gets more than 40:

```
(gdb) break 26 if ii>40
Breakpoint 1 at 0x4009cd: file hello.cpp, line 26.
(gdb) run
Starting program: /home/eijkhout/intro-programming-private/code/gdb/hello
hello world 1
```

```
hello world 2
hello world 5
hello world 10
hello world 17
hello world 26
hello world 37

Breakpoint 1, main () at hello.cpp:26
26          say(ii);
Missing separate debuginfos, use: debuginfo-install glibc-2.17-292.el7.x8
(gdb) print i
$1 = 7
```

37.3.5 A NaN example

The following program:

```
17      float root(float n)
18      {
19        float r;
20        float n1 = n-1.1;
21        r = sqrt(n1);
22        return r;
23      }
24
25      int main() {
26        float x=9,y;
27        for (int i=0; i<20; i++) {
28          y = root(x);
29          cout << "root: " << y << endl;
30          x -= 1.1;
31        }
32
33        return 0;
34      }
```

prints some numbers that are 'not-a-number':

```
[] ./root
root: 2.81069
root: 2.60768
root: 2.38747
root: 2.14476
root: 1.87083
root: 1.54919
root: 1.14018
root: 0.447214
root: -nan
root: -nan
root: -nan
```

Suppose you want to figure out why this happens.

The line that prints the 'nan' is 29, so we want to set a breakpoint there, and preferably a conditional breakpoint. But how do you test on 'nan'? This takes a little trick.

```
(gdb) break 29 if y!=y
Breakpoint 1 at 0x400ea6: file root.cpp, line 28.
(gdb) run
Starting program: /home/eijkhout/intro-programming-private/code/gdb/root
root: 2.81069
root: 2.60768
root: 2.38747
root: 2.14476
root: 1.87083
root: 1.54919
root: 1.14018
root: 0.447214

Breakpoint 1, main () at root.cpp:29
29          cout << "root: " << y << endl;
```

We discover what iteration this happens:

```
(gdb) print i
$1 = 8
```

so now we can rerun the program, and investigate that particular iteration:

```
(gdb) break 28 if i==8
Breakpoint 2 at 0x400eaf: file root.cpp, line 28.
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/eijkhout/intro-programming-private/code/gdb/root
root: 2.81069
root: 2.60768
root: 2.38747
root: 2.14476
root: 1.87083
root: 1.54919
root: 1.14018
root: 0.447214

Breakpoint 2, main () at root.cpp:28
28          y = root(x);
```

We now go into the `root` routine to see what is going wrong there:

```
(gdb) step
root (n=0.200000554) at root.cpp:20
20          float n1 = n-1.1;
(gdb)
21          r = sqrt(n1);
(gdb) print n
$2 = 0.200000554
(gdb) print n1
$3 = -0.89999944
(gdb) next
22          return r;
(gdb) print r
$4 = -nan(0x400000)
```

And there we have the problem: our input `n` is used to compute another number `n1` of which we compute the square root, and sometimes this number gets negative.

37.3.6 Assertions

Instead of running a program and debugging it if you happen to spot a problem (and note that this may not always be the case!) you can also make your program more robust by including *assertions*. These are things that you know should be true, from your knowledge of the problem you are solving.

For instance, in the previous example there was a square root function, and you just 'knew' that the input was always going to be positive. So you edit your program as follows:

```
// header to allow assertions:
#include <cassert>

float root(float n)
{
  float r;
  float n1 = n-1.1;
  assert(n1>=0); // NOTE!
  r = sqrt(n1);
  return r;
}
```

Now if you run your program, you get:

```
[] ./assert
root: 2.81069
root: 2.60768
root: 2.38747
root: 2.14476
root: 1.87083
root: 1.54919
root: 1.14018
root: 0.447214
```

```
assert: assert.cpp:22: float root(float): Assertion 'n1>=0' failed.
Aborted (core dumped)
```

What does this give you?

- It only tells you that an assertion failed, not with what values;
- it does not give you a traceback or so; on the other hand
- assertions can help you detect error conditions that you might otherwise have overlooked!

Chapter 38

Complexity

38.1 Complexity of algorithms

38.1.1 Theory

For the theory of complexity, see HPC book [11], section 19.

38.1.2 Time complexity

Exercise 38.1. For each number n from 1 to 100, print the sum of all numbers 1 through n.

There are several possible solutions to this exercise. Let's assume you don't know the formula for the sum of the numbers $1 \ldots n$. You can have a solution that keeps a running sum, and a solution with an inner loop.

Exercise 38.2. How many operations, as a function of n, are performed in these two solutions?

38.1.3 Space complexity

Exercise 38.3. Read numbers that the user inputs; when the user inputs zero or negative, stop reading. Add up all the positive numbers and print their average.

This exercise can be solved by storing the numbers in a `std::vector`, but one can also keep a running sum and count.

Exercise 38.4. How much space do the two solutions require?

Chapter 39

Support tools

Knowing how to write a program is not enough: you need a variety of tools to be a good programmer, or to be a programmer at all.

39.1 Editors and development environments

Simple programs, such as most of the exercises in this book, can be written using a simple editor. Traditionally, programmers have used *emacs* or *vi* (or one of its derivates such as *vim*).

More sophisticated are development environments such as *Microsoft Visual Studio Code* or *CLion*.

39.2 Compilers

For the simple exercises in this book, all you needed to know about a compiler was the commandline

```
icpx -o myprogram myprogram.cpp
```

(or whatever compiler name and program name you had).

There is much more to know about compilers; see Tutorials book [9], chapter-2.

39.3 Build systems

For programs that are more complicated than a single source file (and even then...) you are wise to use some form of build system.

- The simplest and oldest solution is *Make*; see Tutorials book [9], chapter-3.
- More modern, more powerful, yet also in a way more complicated, is *CMake*; see Tutorials book [9], chapter-4.
- Make and Cmake are often integrated into the above-mentioned development environments.

39.4 Debuggers

If your code misbehaves, having a good *debugger* can be a lifesaver. The traditional debugger is *gdb* (see Tutorials book [9], chapter-11) but, again, this is often integrated into build environments.

Chapter 40

Build systems

Chapter 41

Performance: programming and measurement

41.1 Time measurement

Header

```
#include <chrono>
```

Convenient:

```
using namespace std::chrono
```

but here we spell it all out.

41.1.1 Time durations

You can define durations with `seconds`:

```
seconds s{3};
auto t = 4s;
```

Remark 20 *Use one of the following:*

```
std::chrono::seconds two{2};

using std::chrono;
seconds two{2};

using namespace std::chrono;
seconds two{2};
```

You can do arithmetic and comparisons on this type:

Code:

```
cout << "This lasts "
     << s.count() << "s" << '\n';
cout << "This lasts ";
print_seconds( s+5s );
auto nine = 3.14*3s;
cout << nine.count()
     << "s is under 10 sec: "
     << boolalpha << (nine<10s)
     << '\n';
```

Output
[chrono] basicsecond:

```
This lasts 3s
This lasts 8s
9.42s is under 10 sec: true
```

Note that while *seconds* takes an integer argument, you can then multiply or divide it to get fractional values.

There is a duration *millisecond*, and you can convert seconds implicitly to milli, but the other way around you need *duration_count*:

```
print_milliseconds( 5s );
// DOES NOT COMPILE print_seconds( 6ms );
print_seconds( duration_cast<seconds>(6ms) );
```

The full list of durations (with suffixes) is: *hours* (1*h*), *minutes* (1*min*), *seconds* (1*s*), *milliseconds* (1*ms*), *microseconds* (1*us*), *nanoseconds* (1*ns*).

41.1.2 Time points

A *time point* can be considered as a duration from some starting point, such as the start of the Unix *epoch*: the start of the year 1970.

```
time_point<system_clock,seconds> tp{10'000s};
```

is 2*h*+46*min*+40*s* into 1970.

You make this explicit by calling the *time_since_epoch* method on a time point, giving a duration.

41.1.3 Clocks

There are several clocks. The common supplied clocks are

- *system_clock* for time points that have a relation to the calendar; and
- *steady_clock* for precise measurements.

Usually, *high_resolution_clock* is a synonym for either of these.

A clock has properties:

- *duration*
- *rep*
- *period*
- *time_point*
- *is_steady*
- and a method *now*().

As you saw above, a *time_point* is associated with a clock, and time points of different clocks can not be compared or converted to each other.

41.1.3.1 Duration measurement

To time a segment of execution, use the `now` method of the clock, before and after the segment. Subtracting the time points gives a duration in nanoseconds, which you can cast to anything else:

Code:

```
using clock = system_clock;
clock::time_point  before =
    clock::now();
std::this_thread::sleep_for( 1.5s );
auto after = clock::now();
cout << "Slept for "
     <<
    duration_cast<milliseconds>(after-before).count()
     << "ms\n";
```

Output
[chrono] clock:

```
Slept for 1503ms
```

(The `sleep` function is not a `chrono` function, but comes from the `thread` header; see section **??**.)

41.1.3.2 Clock resolution

The *clock resolution* can be found from the `period` property:

```
auto
num = myclock::period::num,
den = myclock::period::den;
auto tick = static_cast<double>(num)/static_cast<double>(den);
```

Timing:

```
auto start_time = myclock::now();
auto duration = myclock::now()-start_time;
auto microsec_duration =
std::chrono::duration_cast<std::chrono::microseconds>(duration);
cout << "This took " << microsec_duration.count() << "usec" << endl;
```

Computing new time points:

```
auto deadline = myclock.now() + std::chrono::seconds(10);
```

41.1.4 C mechanisms not to use anymore

Letting your process sleep: `sleep`

Time measurement: `getrusage`

PART IV

INDEX AND SUCH

Chapter 42

Index of C++ keywords

Chapter 43

Bibliography

[1] IEEE 754-2019 standard for floating-point arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, 2019. 24

[2] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. LAPACK: A portable linear algebra library for high-performance computers. In *Proceedings Supercomputing '90*, pages 2–11. IEEE Computer Society Press, Los Alamitos, California, 1990. 206

[3] Roy M. Anderson and Robert M. May. Population biology of infectious diseases: Part I. *Nature*, 280:361–367, 1979. doi:10.1038/280361a0. 175

[4] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997. 130

[5] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. PETSc webpage. http://www.mcs.anl.gov/petsc, 2011. 130

[6] Tolga Bektas. The multiple traveling salesman problem: an overview of formulations and solution procedures. *Omega*, 34(3):209 – 219, 2006. 202

[7] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, October 1977. 226

[8] https://github.com/jarro2783/cxxopts. 212

[9] Victor Eijkhout. HPC carpentry. https://theartofhpc.com/carpentry.html. 40, 141, 303

[10] Victor Eijkhout. Parallel programming in mpi and openmp. http://theartofhpc.com/pcse.html. 81, 117

[11] Victor Eijkhout. The science of computing. http://theartofhpc.com/istc.html. 14, 15, 117, 118, 130, 131, 132, 164, 187, 202, 206, 209, 217, 218, 219, 301

[12] D. Goldberg. Compuer arithmetic. Appendix in [15]. 28

[13] Google. Google developer documentation style guide. https://developers.google.com/style/. 141

[14] Kazushige Goto and Robert A. van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.*, 34(3):1–25, 2008. 210

[15] John L. Hennessy and David A. Patterson. *Computer Architecture, A Quantitative Approach*. Morgan Kaufman Publishers, 3rd edition edition, 1990, 3rd edition 2003. 321

[16] https://mathworld.wolfram.com/Kermack-McKendrickModel.html. 175

[17] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw.*, 5(3):308–323, September 1979. 206

[18] Robert Mecklenburg. *Managing Projects with GNU Make*. O'Reilly Media, 3rd edition edition, 2004. Print ISBN:978-0-596-00610-5 ISBN 10:0-596-00610-1 Ebook ISBN:978-0-596-10445-0 ISBN 10:0-596-10445-6. 239

[19] Harry L. Reed. Firing table computations on the eniac. In *Proceedings of the 1952 ACM National Meeting (Pittsburgh)*, ACM '52, page 103?106, New York, NY, USA, 1952. Association for Computing Machinery. 233

[20] Lin S. and Kernighan B. An effective heuristic algorithm for the traveling salesman problem. *Operations Research*, 21:498–516, 1973. 201