C++ course

Victor Eijkhout, Susan Lindsey

Fall 2023

last formatted: February 6, 2024

Basics

Basics

1. Two kinds of files

In programming you have two kinds of files:

- *source files*, which are understandable to you, and which you create with an editor such as `vi` or `emacs`; and
- *binary files*, which are understandable to the computer, and unreadable to you.

Your source files are translated to binary by a compiler, which 'compiles' your source file.

Exercise 1

Make a file `zero.cc` with the following lines:

```
// basic/null.cpp
#include <iostream>
using std::cout;

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

and compile it. Intel compiler:

```
icpc -o zeroprogram zero.cc
```

Run this program (it gives no output):

```
./zeroprogram
```

2. Anatomy of the compile line

- `icpc` : compiler. Alternative: use `g++` or `clang++`
- `-o zeroprogram` : output into a binary name of your choosing
- `zero.cc` : your source file.

Exercise 2

Add this line:

```
// basic/hello.cpp
cout << "Hello world!" << '\n';
```

(copying from the pdf file is dangerous! please type it yourself)

Compile and run again. What is the output?

3. C++ versions

- The compiler by default uses `C++98`.
- This course explains `C++17` and `C++20`. You need tell your compiler about this.
- On `isp.tacc.utexas.edu` 'icpc' uses this by default.
- On your own machine you need to do

  ```
  g++ -std=c++17 [other options]
  ```

  or

  ```
  alias g++='g++ -std=c++17'
  ```

  (in your `.bashrc` file)

Review quiz 1

True or false?

1. The programmer only writes source files, no binaries.
   `/poll "A programmer writes sources, no binaries" "T" "F"`
2. The computer only executes binary files, no human-readable files.
   `/poll "Computer executes binaries, no readable files" "T" "F"`

4. Error types

Your code may suffer from the following types of error:

1. *Syntax* or *compile-time* errors: these arise if what you write is not according to the language specification. The compiler catches these errors, and it refuses to produce a binary file.
2. *Run-time* errors: these arise if your code is syntactically correct, and the compiler has produced an executable, but the program does not behave the way you intended or foresaw. Examples are divide-by-zero or indexing outside the bounds of an array.
3. Design errors: your program does not do what you think it does.

5. File names

File names can have extensions: the part after the dot. (The part before the dot is completely up to you.)

- `program.cpp` or `program.cc` or `program.cxx` are typical extensions for C++ sources.
- `program.cpp` has a possible possible confusion with 'C PreProcessor', but it seems to be the standard, so we will use it in this course.
- Using `program` without extension usually indicates an executable. (What happens if you leave the `-o myprogram` part off the compile line?)

Statements

6. Program statements

- A program contains statements, each terminated by a semicolon.
- 'Curly braces' can enclose multiple statements.
- A statement can correspond to some action when the program is executed.
- Some statements are definitions, of data or of possible actions.
- Comments are 'Note to self', short:

```
cout << "Hello world" << '\n'; // say hi!
```

and arbitrary:

```
cout << /* we are now going
            to say hello
         */ "Hello!" << /* with newline: */ '\n';
```

Exercise 3

Take the 'hello world' program you wrote above, and duplicate the hello-line. Compile and run.

Does it make a difference whether you have the two hellos on the same line of your file, or on different lines?

Experiment with other changes to the layout of your source. Find at least one change that leads to a compiler error. Can you relate the message to the error?

7. Fixed elements

You see that certain parts of your program are inviolable:

- There are keywords such as `return` or `cout`; you can not change their definition.
- Curly braces and parentheses need to be matched.
- There has to be a *main* keyword.
- The *iostream* and `std` are usually needed.

Exercise 4

Experiment with the `cout` statement. Replace the string by a number or a mathematical expression. Can you guess how to print more than one thing, for instance:

- the string `One third is`, and
- the result of the computation $1/3$,

with the same `cout` statement? Do you get anything unexpected?

Review quiz 2

True or false?

- If your program compiles correctly, it is correct.
  `/poll "If it compiles it is correct" "T" "F"`
- If you run your program and you get the right output, it is correct.
  `/poll "Program runs, no errors, therefore it is correct" "T" "F"`

Variables

8. What's a variable?

Programs usually contain data, which is stored in a variable.
A variable has

- a datatype,
- a name, and
- a value.

These are defined in a variable declaration and/or variable assignment.

9. Example variable lifetime

```
int i,j; // declaration
i = 5;   // set a value
i = 6;   // set a new value
j = i+1; // use the value of i
i = 8; // change the value of i
       // but this doesn't affect j:
       // it is still 7.
```

10. Variable names

- A variable name has to start with a letter;
- a name can contains letters and digits, but not most special characters, except for the underscore.
- For letters it matters whether you use upper or lowercase: the language is case sensitive.
- Words such as `main` or `return` are reserved words.
- Usually `i` and `j` are not the best variable names: use `row` and `column`, or other meaningful names, instead.
- While you can start a name with an underscore, there are some limitations on the use of the underscore: do not use two underscores in a row, and do not start a name with an underscore followed by a capital letter.

11. Declaration

A variable declaration establishes the name and the type of a variable:

```
int n_elements;
float value;
int row,col;
double re_part,im_part;
```

12. Where do declarations go?

Declarations can go pretty much anywhere in your program, but they need to come before the first use of the variable.

Note: it is legal to define a variable before the main program but such global variables are usually not a good idea. Please only declare variables *inside* main (or inside a function et cetera).

Review quiz 3

Which of the following are legal variable names?

1. `mainprogram`
   `/poll "Legal mainprogram?" "T" "F"`
2. `main`
   `/poll "Legal main?" "T" "F"`
3. `Main`
   `/poll "Legal Main?" "T" "F"`
4. `1forall`
   `/poll "Legal 1forall?" "T" "F"`
5. `one4all`
   `/poll "Legal one4all?" "T" "F"`
6. `one_for_all`
   `/poll "Legal one_for_all?" "T" "F"`
7. `onefor{all}`
   `/poll "Legal onefor{all}?" "T" "F"`

13. Datatypes

Variables come in different types;

- We call a variable of type `int`, `float`, `double` a numerical variable.
- *Complex numbers* will be discussed later.
- For characters: `char`. Strings are complicated; see later.
- Truth values: `bool`
- You can make your own types. Later.

Assignments

14. Assignment

Once you have declared a variable, you need to establish a value. This is done in an assignment statement. After the above declarations, the following are legitimate assignments:

```
n = 3;
x = 1.5 - n;
n1 = 7;
n2 = n1 * 3;
```

These are not math equations: the variable on the left hand side gets the value of the expression on the right hand side.

You see that you can assign both a simple value or an expression.

15. Special forms

Certain assignments with the same variable in both the left and right hand sides can be simplified:

```
x = x+2; y = y/3;
// can be written as
x += 2; y /= 3;
```

Integer add/subtract one:

```
i=i+1; j=j-1;
// rewritten as:
++i; --j;
// or
i++; j--;
```

Review quiz 4

Which of the following are legal? If they are, what is their meaning?

1. `n = n;`
   ```
   /poll "Legal 'n = n;'?" "T" "F"
   ```
2. `n = 2n;`
   ```
   /poll "Legal 'n = 2n;'?" "T" "F"
   ```
3. `n = n2;`
   ```
   /poll "Legal 'n = n2;'?" "T" "F"
   ```
4. `n = 2*k;`
   ```
   /poll "Legal 'n = 2*k;'?" "T" "F"
   ```
5. `n/2 = k;`
   ```
   /poll "Legal 'n/2 = k;'?" "T" "F"
   ```
6. `n /= k;`
   ```
   /poll "Legal 'n /= k;'?" "T" "F"
   ```

16. Initialization syntax

There are (at least) two ways of initializing a variable

```
int i = 5;
int j{6};
```

Note that writing

```
int i;
i = 7;
```

is not an initialization: it's a declaration followed by an assignment.

Review quiz 5

```
#include <iostream>
using std::cout;
int main() {
  int i;
  int j = i+1;
  cout << j << "\n";
  return 0;
}
```

What happens?

1. Compiler error
2. Output: `1`
3. Output is undefined
4. Error message during running the program.

```
/poll "What happens:" "Compiler error" "Output: 1" "Output undefined" "Runtime error"
```

17. Floating point constants

- Default: `double`
- Float: `3.14f` or `3.14F`
- Long double: `1.22l` or `1.22L`.

18. Warning: floating point arithmetic

Floating point arithmetic is full of pitfalls.

- Don't count on `3*(1./3)` being exactly 1.
- Not even associative.

19. Truth values

So far you have seen integer and real variables. There are also boolean values which represent truth values. There are only two values: `true` and `false`.

```
bool found{false};
found = true;
```

Input/Output

20. Terminal output

Terminal (console) output with `cout`:

```
float x = 5;
cout << "Here is the root: " << sqrt(x) << '\n';
```

Note the newline character.
Alternatively: `std::endl`, less efficient.

21. Terminal input

```
// basic/cin.cpp
string name; int age;
cout << "Your name?\n";
cin >> name;
cout << "age?\n";
cin >> age;
cout << age << " is a nice
    age, "
     << name << '\n';
```

```
> ./cin
Your name?
Victor
age?
18
18 is a nice age, Victor
> ./cin
Your name?
THX 1138
age?
1138 is a nice age, THX
```

22. Quick intro to strings

- Add the following at the top of your file:

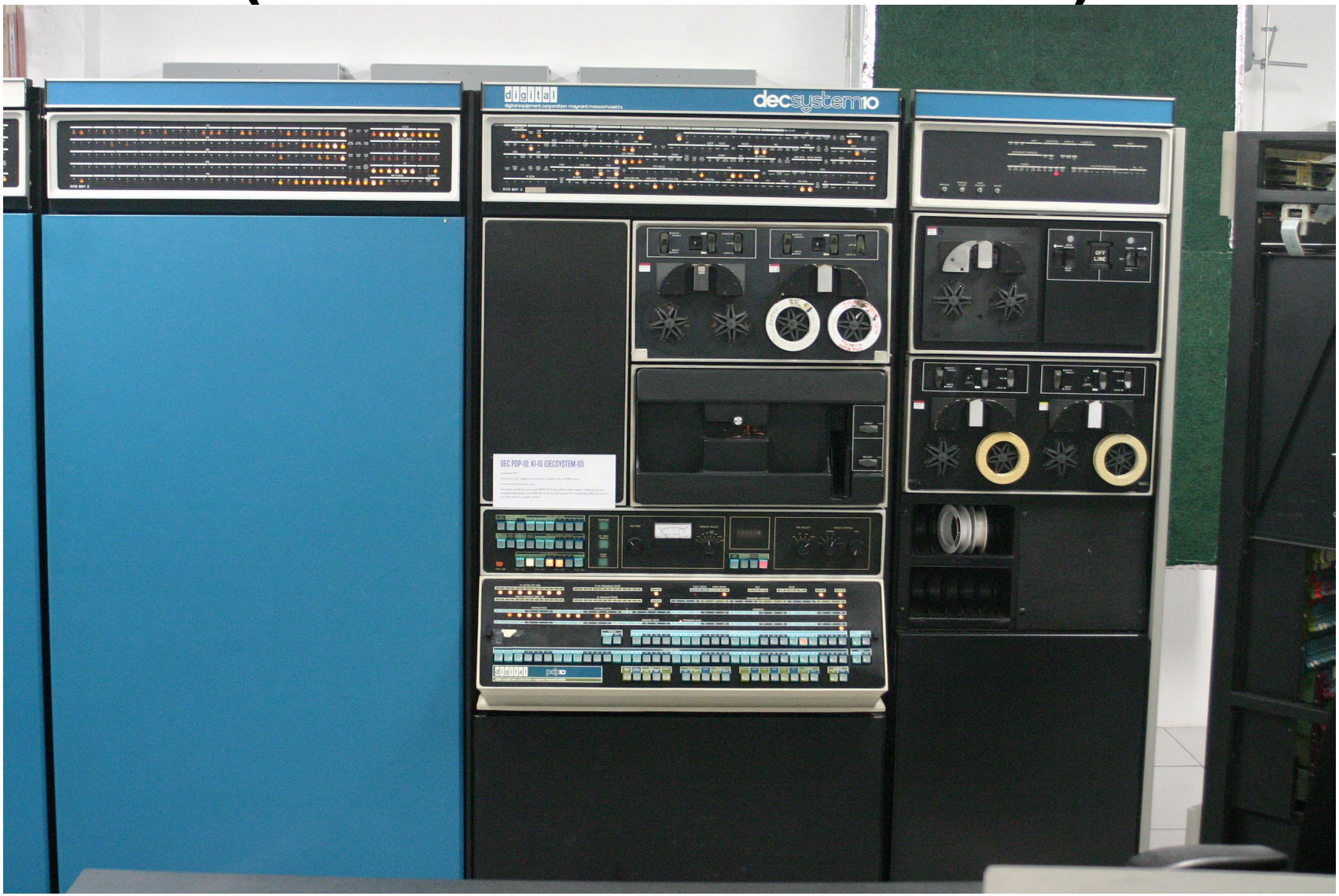
```
#include <string>
using std::string;
```

- Declare string variables as

```
string name;
```

- And you can now `cin` and `cout` them.

(Just what *is* a console?)



Exercise 5

Write a program that asks for the user's first name, uses `cin` to read that, and prints something like `Hello, Susan!` in response.

What happens if you enter first and last name?

Expressions

23. Arithmetic expressions

- Expression looks pretty much like in math.
  With integers: `2+3`
  with reals: `3.2/7`
- Use parentheses to group `25.1*(37+42/3.)`
- Careful with types.
- There is no 'power' operator: library functions.
- Modulus: `%`

24. Math library calls

Math functions are in `cmath`:

```
#include <cmath>
.....
x = pow(3,.5);
```

For squaring, usually better to write `x*x` than `pow(x,2)`.

Boolean expressions

We'll do that in the lecture on conditionals.

Exercise 6

Write a program that :

- displays the message `Type a number`,
- accepts an integer number from you (use `cin`),
- makes another variable that is three times that integer plus one,
- and then prints out the second variable.

Turn it in!

- If you have compiled your program, do:

  `coe_3np1 yourprogram.cc`

  where '`yourprogram.cc`' stands for the name of your source file.
- Is it reporting that your program is correct? If so, do:

  `coe_3np1 -s yourprogram.cc`

  where the `-s` flag stands for 'submit'.

Note: this will send your file to the instructors with a **time stamp**. If you submit again after the deadline, you will be recorded as a late submission.

25. Conversion and casting

Real to integer: round down:

```
double x,y; x = .... ; y = .... ;
int i; i =  x+y:
```

Dangerous:

```
int i,j; i = ... ; j = ... ;
double x ; x = 1+i/j;
```

The fraction is executed as integer division.
For floating point result do:

```
(1.*i)/j
```

Optional exercise 7

Write a program that asks for two integer numbers `n1,n2`.

- Assign the integer ratio n_1/n_2 to an integer variable.
- Can you use this variable to compute the modulus

$$n_1 \mod n_2$$

  (without using the % modulus operator!)
  Print out the value you get.
- Also print out the result from using the modulus operator:%.
- Investigate the behavior of your program for negative inputs.
  Do you get what you were expecting?

Optional exercise 8

Write two programs, one that reads a temperature in Centigrade and converts to Fahrenheit, and one that does the opposite conversion.

$$C = (F - 32) \cdot 5/9, \qquad F = 9/5\, C + 32$$

Check your program for the freezing and boiling point of water. (Do you know the temperature where Celsius and Fahrenheit are the same?)

Can you use Unix pipes to make one accept the output of the other?

Review quiz 6

True of false?

1. Within a certain range, all integers are available as values of an integer variable.
2. Within a certain range, all real numbers are available as values of a float variable.
3. `5(7+2)` is equivalent to 45.
4. `1--1` is equivalent to zero.
5. `int i = 5/3.;` The variable `i` is 2.
6. `float x = 2/3;` The variable `x` is approximately `0.6667`.

Control structures

Reference material

The following slides are a high-level introduction;
for details see: chapter Textbook, section 5 upto sectionTextbook, section 5.5

26. If-then-else

A conditional is a test: 'if something is true, then do this, otherwise maybe do something else'. The C++ syntax is

```
if ( something ) {
  // do something;
} else {
  // do otherwise;
}
```

- The 'else' part is optional
- You can leave out braces in case of single statement.

27. Complicated conditionals

Chain:

```
if ( /* some test */ ) {
  ...
} else if ( /* other test */ ) {
  ...
}
```

Nest:

```
if ( /* some test */ ) {
  if ( /* other test */ ) {
    ...
  } else {
    ...
  }
}
```

28. Comparison and logical operators

Here are the most common logic operators and comparison operators:

Operator	meaning	example
==	equals	`x==y-1`
!=	not equals	`x*x!=5`
>	greater	`y>x-1`
>=	greater or equal	`sqrt(y)>=7`
<,<=	less, less equal	
&&,||	and, or	`x<1 && x>0`
and,or	and, or	`x<1 and x>0`
!	not	`!( x>1 && x<2 )`
not		`not ( x>1 and x<2 )`

Precedence rules of operators are common sense. When in doubt, use parentheses.

Exercise 9

The following code claims to detect if an integer has more than 2 digits.

Code:

```
1 // basic/if.cpp
2 int i;
3 cin >> i;
4 if ( i>100 )
5   cout << "That number " << i
6        << " had more than 2 digits"
7        << '\n';
```

Output:

```
... with 50 as input
     ....
... with 150 as input
     ....
That number 150 had
     more than 2 digits
```

Fix the small error in this code. Also add an 'else' part that prints if a number is negative.

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

Review quiz 7

True or false?

- The tests `if (i>0)` and `if (0<i)` are equivalent.
  `/poll "Same tests: 'i>0' and '0<i' ?" "T" "F"`
- The test

  ```
  if (i<0 && i>1)
    cout << "foo"
  ```

  prints `foo` if $i < 0$ and also if $i > 1$.
  `/poll "'if (i<0 && i>1)' is true if i negative and if i greater than one" "T" "F"`
- The test

  ```
  if (0<i<1)
    cout << "foo"
  ```

  prints `foo` if i is between zero and one.
  `/poll "'if (0<i<1)' true if i between 0 and 1" "T" "F"`

Review quiz 8

Any comments on the following?

```
bool x;
// ... code with x ...
if ( x == true )
  // do something
```

Exercise 10

Read in an integer. If it is even, print ‘even’, otherwise print ’odd’:

```
if ( /* your test here */ )
  cout << "even" << '\n';
else
  cout << "odd" << '\n';
```

Then, rewrite your test so that the true branch corresponds to the odd case.

Exercise 11

Read in a positive integer. If it's a multiple of three print 'Fizz!'; if it's a multiple of five print 'Buzz!'. It is a multiple of both three and five print 'Fizzbuzz!'. Otherwise print nothing.

Note:

- Capitalization.
- Exclamation mark.
- Your program should display at most one line of output.

Turn it in!

- If you have compiled your program, do:

  `coe_fizzbuzz yourprogram.cc`

  where '`yourprogram.cc`' stands for the name of your source file.
- Is it reporting that your program is correct? If so, do:

  `coe_fizzbuzz -s yourprogram.cc`

  where the `-s` flag stands for 'submit'.

Note: this will send your file to the instructors with a **time stamp**. If you submit again after the deadline, you will be recorded as a late submission.

Prime Project Exercise 12

Read two numbers and print a message stating whether the second is as divisor of the first:

Code:

```
1 // primes/divisiontest.cpp
2   int number,divisor;
3   bool is_a_divisor;
4     /* ... */
5   if (
6     /* ... */
7       ) {
8     cout << "Indeed, " << divisor
9          << " is a divisor of "
10         << number << '\n';
11  } else {
12    cout << "No, " << divisor
13         << " is not a divisor of "
14         << number << '\n';
15  }
```

Output:

```
( 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
```

29. Switch statement example

Cases are executed consecutively until you 'break' out of the switch statement:

Code:

```
1 // basic/switch.cpp
2 switch (n) {
3 case 1 :
4 case 2 :
5   cout << "very small" << '\n';
6   break;
7 case 3 :
8   cout << "trinity" << '\n';
9   break;
10 default :
11   cout << "large" << '\n';
12 }
```

Output:

```
for v in 1 2 3 4 5 ; do \
          echo $v | ./switch
    ; \
       done
very small
very small
trinity
large
large
```

30. Local variables in conditionals

The curly brackets in a conditional allow you to define local variables:

```
if ( something ) {
  int i;
  .... do something with i
}
// the variable `i' has gone away.
```

Good practice: only define variable where needed.

Braces induce a scope.

Reference material

The following slides are a high-level introduction;
for details see: chapter Textbook, section 6 upto sectionTextbook, section 6.4

For loops

31. 'For' statement

Sometimes you need to repeat a statement a number of times. That's where the loop comes in. A loop has a counter, called a loop variable, which (usually) ranges from a lower bound to an upper bound.

Here is the syntax in the simplest case:

```
// loop/sumsquares.cpp
int sum_of_squares{0};
for (int var=low; var<upper; ++var) {
  sum_of_squares += var*var;
}
cout << "The sum of squares from "
     << low << " to " << upper
     << " is " << sum_of_squares << '\n';
```

32. Loop syntax: variable

The loop variable is usually an integer:

```
for ( int index=0; index<max_index; index=index+1) {
  ...
}
```

But other types are allowed too:

```
for ( float x=0.0; x<10.0; x+=delta ) {
  ...
}
```

Beware the stopping test for non-integral variables!

33. Loop syntax: test

- If this boolean expression is true, do the next iteration.
- Done before the first iteration too!
- Test can be empty. This means no test is applied.

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

34. Loop syntax: increment

Increment performed after each iteration. Most common:

- `i++` for a loop that counts forward;
- `i--` for a loop that counts backward;

Others:

- `i+=2` to cover only odd or even numbers, depending on where you started;
- `i*=10` to cover powers of ten.

Even optional:

```
for (int i=0; i<N; ) {
  // stuff
  if ( something ) i+=1; else i+=2;
}
```

Review quiz 9

For each of the following loop headers, how many times is the body executed? (You can assume that the body does not change the loop variable.)

```
for (int i=0; i<7; i++)
```

```
/poll "for (int i=0; i<7; i++)" "6 iterations" "7" "8"
```

```
for (int i=0; i<=7; i++)
```

```
/poll "for (int i=0; i<=7; i++)" "6 iterations" "7" "8"
```

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

```
/poll "for (int i=0; i<0; i++)" "0 iterations" "1" "inf"
```

Review quiz 10

What is the last iteration executed?

```
for (int i=1; i<=2; i=i+2)
```

```
/poll "for (int i=1; i<=2; i=i+2) last iteration" "i=1" "i=2" "i=3" "i=4"
```

```
for (int i=1; i<=5; i*=2)
```

```
/poll "for (int i=1; i<=5; i*=2) last iteration" "4" "5" "8"
```

```
for (int i=0; i<0; i--)
```

```
/poll "for (int i=0; i<0; i--) last iteration" "none" "0" "-1" "-inf"
```

```
for (int i=5; i>=0; i--)
```

```
/poll "for (int i=5; i>=0; i--) last iteration" "0" "1" "-1" "4"
```

```
for (int i=5; i>0; i--)
```

Exercise 13

Take this code:

```
// loop/sumsquares.cpp
int sum_of_squares{0};
for (int var=low; var<upper; ++var) {
  sum_of_squares += var*var;
}
cout << "The sum of squares from "
     << low << " to " << upper
     << " is " << sum_of_squares << '\n';
```

and modify it to sum only the squares of every other number, starting at *low*.

Can you find a way to sum the squares of the even numbers $\geq$*low*?

Programming Project Exercise 14

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.

35. Nested loops

Traversing a matrix
(we will discuss actual matrix data structures later):

```
for (int row=0; row<m; row++)
  for (int col=0; col<n; col++)
    ...
```

This is called 'loop nest', with
row: outer loop
col: inner loop.

36. Indefinite looping

Sometimes you want to iterate some statements not a predetermined number of times, but until a certain condition is met. There are two ways to do this.

First of all, you can use a 'for' loop and leave the upper bound unspecified:

```
for (int var=low; ; var=var+1) { ... }
```

37. Break out of a loop

This loop would run forever, so you need a different way to end it. For this, use the `break` statement:

```
for (int var=low; ; var=var+1) {
  statement;
  if (some_test) break;
  statement;
}
```

Exercise 15

The integer sequence

$$u_{n+1} = \begin{cases} u_n/2 & \text{if } u_n \text{ is even} \\ 3u_n + 1 & \text{if } u_n \text{ is odd} \end{cases}$$

leads to the Collatz conjecture: no matter the starting guess u_1, the sequence $n \mapsto u_n$ will always terminate at 1.

$$5 \to 16 \to 8 \to 4 \to 2 \to 1$$

$$7 \to 22 \to 11 \to 34 \to 17 \to 52 \to 26 \to 13 \to 40 \to 20 \to 10 \to 5 \cdots$$

(What happens if you keep iterating after reaching 1?)

Try all starting values $u_1 = 1, \ldots, 1000$ to find the values that lead to the longest sequence: every time you find a sequence that is longer than the previous maximum, print out the starting number.

Breaking out of a loop

Reference material

The following slides are a high-level introduction;
for details see: section Textbook, section 6.3

38. Where did the break happen?

Suppose you want to know what the loop variable was when the break happened. You need the loop variable to be global:

```
int var;
... code that sets var ...
for ( ; var<upper; var++) {
  ... statements ...
  if (some condition) break
  ... more statements ...
}
... code that uses the breaking value of var ...
```

In other cases: define the loop variable in the header!

39. Test in the loop header

If the test comes at the start or end of an iteration, you can move it to the loop header:

```
bool need_to_stop{false};
for (int var=low; !need_to_stop ; var++) {
  ... some code ...
  if ( some condition )
    need_to_stop = true;
}
```

Exercise 16

Write an i, j loop nest that prints out all pairs with

$$1 \leq i, j \leq 10, \quad j \leq i.$$

Output one line for each `i` value.

Now write an i, j loop that prints all pairs with

$$1 \leq i, j \leq 10, \quad |i - j| < 2,$$

again printing one line per `i` value. Food for thought: this exercise is definitely easiest with a conditional in the inner loop, but can you do it without?

Optional exercise 17

Find all triples of integers u, v, w under 100 such that $u^2 + v^2 = w^2$.
Make sure you omit duplicates of solutions you have already found.

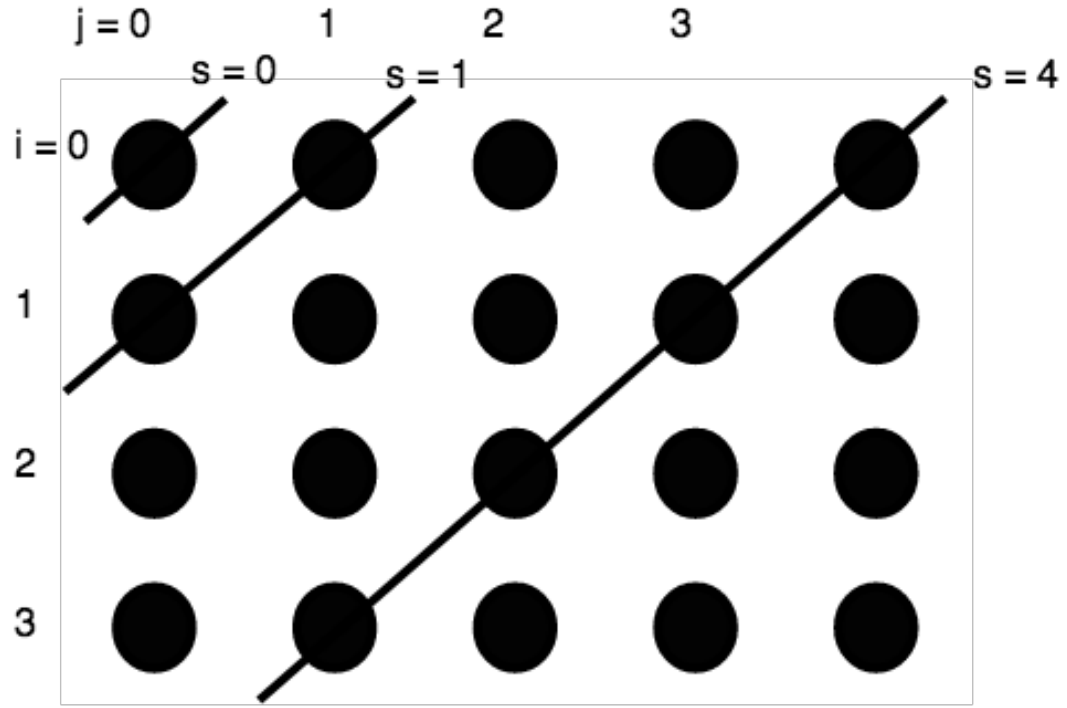
Exercise 18

Write a double loop over $0 \leq i, j < 10$ that prints the first pair where the product of indices satisfies $i \cdot j > N$, where N is a number your read in. A good test case is $N = 40$.

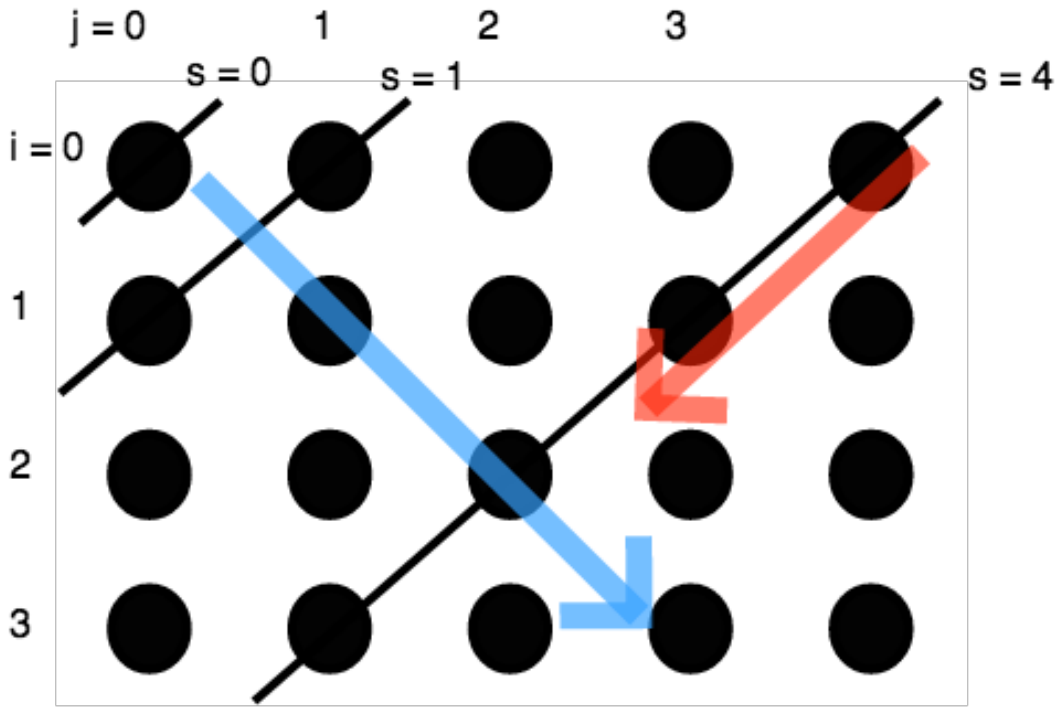
Secondly, find a pair with $i \cdot j > N$, but with the smallest value for $i + j$. (If there is more than one pair, report the one with lower i value.) Can you traverse the i, j indices such that they first enumerate all pairs $i + j = 1$, then $i + j = 2$, then $i + j = 3$ et cetera? Hint: write a loop over the sum value $1, 2, 3, \ldots$, then find i, j.

You program should print out both pairs, each on a separate line, with the numbers separated with a comma, for instance `8,5`.

Suggestive picture 1



Suggestive picture 2



Turn it in!

- If you have compiled your program, do:

  `coe_ij yourprogram.cc`

  where '`yourprogram.cc`' stands for the name of your source file.
- Is it reporting that your program is correct? If so, do:

  `coe_ij -s yourprogram.cc`

  where the `-s` flag stands for 'submit'.

40. Skip iteration

```
for (int var=low; var<N; var++) {
  statement;
  if (some_test) {
    statement;
    statement;
  }
}
```

Alternative:

```
for (int var=low; var<N; var++) {
  statement;
  if (!some_test) continue;
  statement;
  statement;
}
```

The only difference is in layout.

While loops

Reference material

The following slides are a high-level introduction;
for details see: section Textbook, section 6.3.1

41. While loop

Syntax:

```
while ( condition ) {
  statements;
}
```

or

```
do {
  statements;
} while ( condition );
```

42. Pre-test while loop

```
float money = inheritance();
while ( money < 1.e+6 )
  money += on_year_savings();
```

43. While syntax 1

Code:

```
1 // basic/whiledo.cpp
2 cout << "Enter a positive number: "
      ;
3 cin >> invar; cout << '\n';
4 cout << "You said: " << invar <<
      '\n';
5 while (invar<=0) {
6   cout << "Enter a positive number:
      " ;
7   cin >> invar; cout << '\n';
8   cout << "You said: " << invar <<
      '\n';
9 }
10 cout << "Your positive number was "
11      << invar << '\n';
```

Output:

```
Enter a positive number:
You said: -3
Enter a positive number:
You said: 0
Enter a positive number:
You said: 2
Your positive number
     was 2
```

Problem: code duplication.

44. While syntax 2

Code:

```
1 // basic/dowhile.cpp
2 int invar;
3 do {
4   cout << "Enter a positive number:
      " ;
5   cin >> invar; cout << '\n';
6   cout << "You said: " << invar <<
      '\n';
7 } while (invar<=0);
8 cout << "Your positive number was: "
9      << invar << '\n';
```

Output:

```
Enter a positive number:
You said: -3
Enter a positive number:
You said: 0
Enter a positive number:
You said: 2
Your positive number
     was: 2
```

The post-test syntax leads to more elegant code.

Optional exercise 19

A horse is tied to a post with a 1 meter elastic band. A spider that was sitting on the post starts walking to the horse over the band, at 1cm/sec. This startles the horse, which runs away at 1m/sec. Assuming that the elastic band is infinitely stretchable, will the spider ever reach the horse?

Functions

Reference material

The following slides are a high-level introduction;
for details see: chater Textbook, section 7

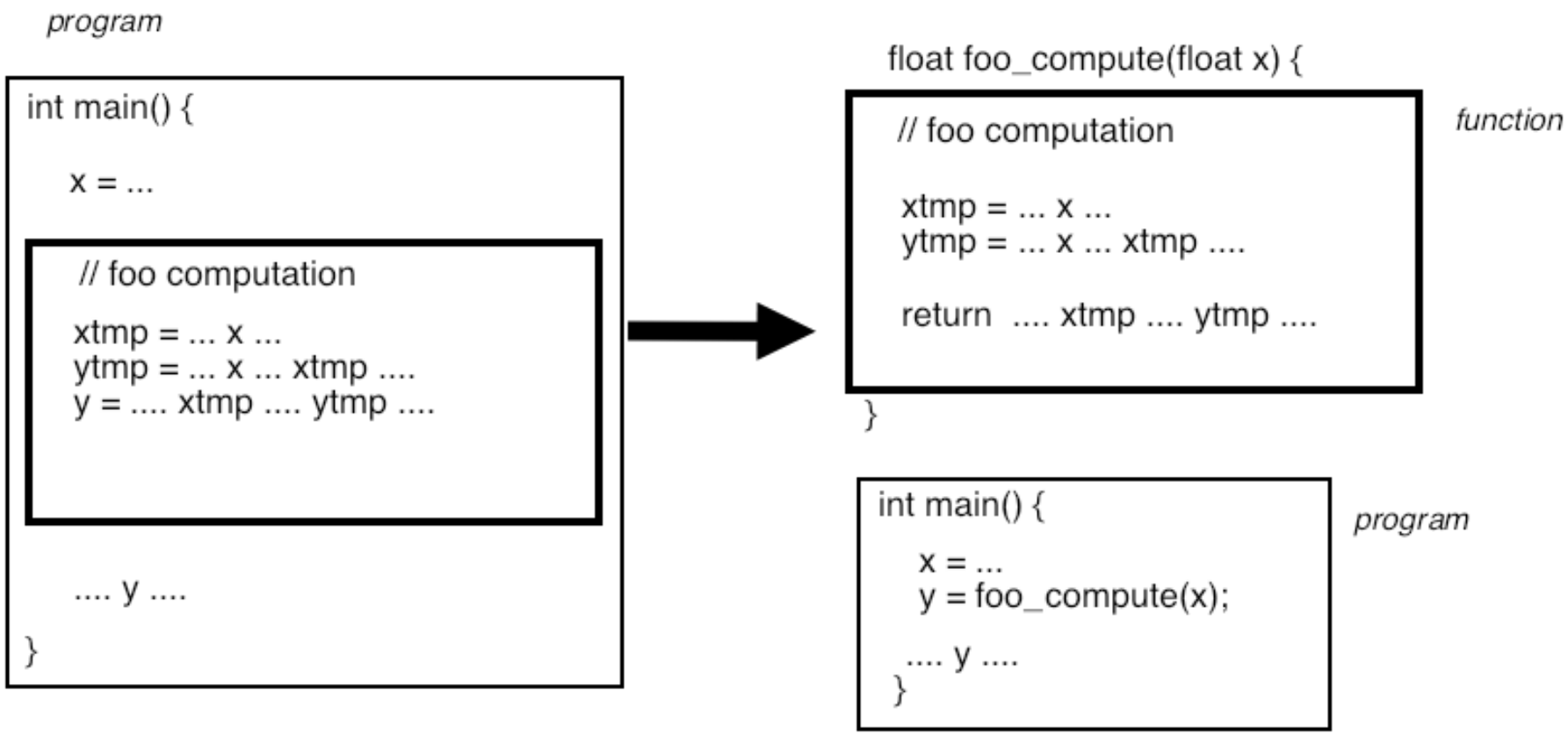
Function basics

45. Why functions?

Functions are an abstraction mechanism.

- Code fragment with clear function:
- Turn into *subprogram*: function *definition*.
- Use by single line: function *call*.
- Abstraction: you have introduced a **name** for a section of code.

46. Introducing a function



47. Program without functions

Example: zero-finding through bisection.

$$\underset{x}{?}: f(x) = 0, \qquad f(x) = x^3 - x^2 - 1$$

(where the question mark quantor stands for 'for which x').

First attempt at coding this: everything in the main program.

Code:

```
1 // func/bisect1.cpp
2 float left{0.},right{2.},
3   mid;
4 while (right-left>.1) {
5   mid = (left+right)/2.;
6   float fmid =
7     mid*mid*mid - mid*mid-1;
8   if (fmid<0)
9     left = mid;
10  else
11    right = mid;
12 }
13 cout << "Zero happens at: " << mid
   << '\n';
```

Output:

```
Zero happens at: 1.4375
```

48. Introducing functions, step 1

Introduce a function for the expression `m*m*m - m*m-1`:

```
// func/bisect2.cpp
float f(float x) {
  return x*x*x - x*x-1;
};
```

Used in main:

```
// func/bisect2.cpp
while (right-left>.1) {
  mid = (left+right)/2.;
  float fmid = f(mid);
  if (fmid<0)
    left = mid;
  else
    right = mid;
}
```

49. Introducing functions, step 2

Function:

```
// func/bisect3.cpp
float f(float x) {
  return x*x*x - x*x-1;
};
float find_zero_between
    (float l,float r) {
  float mid;
  while (r-l>.1) {
    mid = (l+r)/2.;
    float fmid = f(mid);
    if (fmid<0)
      l = mid;
    else
      r = mid;
  }
  return mid;
};
```

New main:

```
// func/bisect3.cpp
int main() {
  float left{0.},right{2.};
  float zero =

    find_zero_between(left,right);
  cout << "Zero happens at: "
       << zero << '\n';
  return 0;
}
```

Exercise 20

Make the bisection algorithm more elegant by introducing functions `new_l`, `new_r` used as:

```
l = new_l(l,mid,fmid);
r = new_r(r,mid,fmid);
```

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

Question: you could leave out `fmid` from the functions. Write this variant. Why is this not a good idea?

50. Why functions?

- Easier to read: use application terminology
- Shorter code: reuse
- Cleaner code: local variables are no longer in the main program.
- Maintainance and debugging

51. Code reuse

Suppose you do the same computation twice:

```
double x,y, v,w;
y = ..... computation from x .....
w = ..... same computation, but from v .....
```

With a function this can be replaced by:

```
double computation(double in) {
  return .... computation from `in' ....
}

y = computation(x);
w = computation(v);
```

52. Code reuse

Example: multiple norm calculations:

Repeated code:

```
float s = 0;
for (int i=0; i<x.size(); i++)
  s += abs(x[i]);
cout << "One norm x: " << s <<
    endl;
s = 0;
for (int i=0; i<y.size(); i++)
  s += abs(y[i]);
cout << "One norm y: " << s <<
    endl;
```

becomes:

```
float OneNorm( vector<float> a
    ) {
  float sum = 0;
  for (int i=0; i<a.size();
    i++)
    sum += abs(a[i]);
  return sum;
}
int main() {
  ... // stuff
  cout << "One norm x: "
       << OneNorm(x) << endl;
  cout << "One norm y: "
       << OneNorm(y) << endl;
```

(Don't worry about array stuff in this example)

Review quiz 11

True or false?

- The purpose of functions is to make your code shorter.
  `/poll "Functions are to make your code shorter" "T" "F"`
- Using functions makes your code easier to read and understand.
  `/poll "Functions make your code easier to understand" "T" "F"`
- Functions have to be defined before you can use them.
  `/poll "Functions have to be defined before use" "T" "F"`
- Function definitions can go inside or outside the main program.
  `/poll "Function definitions can go in or out main" "T" "F"`

53. Declaration first, definition last

Some people like the following style of defining a function:

```
// declaration before main
int my_computation(int);

int main() {
  int result;
  result = my_computation(5);
  return 0;
};

// definition after main
int my_computation(int i) {
   return i+3;
}
```

This is purely a matter of style.

54. Anatomy of a function definition

```
void write_to_file(int i,double x) { /* ... */ }
float euler_phi(int i,bool tf) { /* ... */ return x; }
```

- Result type: what's computed.
  `void` if no result
- Name: make it descriptive.
- Parameters: zero or more.
  `int i,double x,double y`
  These act like variable declarations.
- Body: any length. This is a scope.
- Return statement: usually at the end, but can be anywhere; the computed result. Not necessary for a `void` function.

55. Function call

The function call

1. copies the value of the function argument to the function parameter;
2. causes the function body to be executed, and
3. the function call is replaced by whatever you `return`.
4. (If the function does not return anything, for instance because it only prints output, you declare the return type to be `void`.)

Review quiz 12

True or false?

- A function can have only one input
  `/poll "Function can have only one input" "T" "F"`
- A function can have only one return result
  `/poll "Function can have only one return result" "T" "F"`
- A void function can not have a `return` statement.
  `/poll "Void function can not have 'return'" "T" "F"`

Exercise 21

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: $1, 0$; $0, 1$; $1, 1$; $3, 4$.

Programming Project Exercise 22

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 Textbook, section 12.3.3.)

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?

Programming Project Exercise 23

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.)

Turn it in!

- If you have compiled your program, do:

  `coe_primes yourprogram.cc`

  where '`yourprogram.cc`' stands for the name of your source file.

- Is it reporting that your program is correct? If so, do:

  `coe_primes -s yourprogram.cc`

  where the `-s` flag stands for 'submit'.

- If you don't manage to get your code working correctly, you can submit as incomplete with

  `coe_primes -i yourprogram.cc`

- (Like all good unix programs, the tester also accepts a `-h` flag for 'help'.)

56. Background: square roots by Newton's

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 $f_y(x) \approx 0$.

Optional exercise 24

Compute $\sqrt{2}$ as the zero of $f_y(x) = x^2 - y$ for the special case of $y = 2$.

- Write functions `f(x)` and `deriv(x)`, that compute $f_y(x)$ and $f'_y(x)$ for the particular definition of f_y.
- Iterate until $|f(x, y)| < 10^{-5}$. Print x and $f(x)$ in each iteration; don't worry too much about the stopping test and accuracy attained.
- Second part: write a function `newton_root` that computes $\sqrt{y}$ again: only for $\sqrt{2}$.

Parameter passing

Reference material

The following slides are a high-level introduction;
for details see: section Textbook, section 7.5

57. Mathematical type function

Pretty good design:

- pass data into a function,
- return result through `return` statement.
- Parameters are copied into the function. (Cost of copying?)
- pass by value
- 'functional programming'

58. Pass by value example

Note that the function alters its parameter x:

Code:

```
1 // func/passvalue.cpp
2 double squared( double x ) {
3   double y = x*x;
4   return y;
5 }
6     /* ... */
7   number = 5.1;
8   cout << "Input starts as: "
9        << number << '\n';
10  other = squared(number);
11  cout << "Output var is: "
12       << other << '\n';
13  cout << "Input var is now: "
14       << number << '\n';
```

Output:

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

but the argument in the main program is not affected.

59. Reference

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

Code:

```
1 // basic/ref.cpp
2 int i;
3 int &ri = i;
4 i = 5;
5 cout << i << "," << ri << '\n';
6 i *= 2;
7 cout << i << "," << ri << '\n';
8 ri -= 3;
9 cout << i << "," << ri << '\n';
```

Output:

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

(You will not use references often this way.)

60. Parameter passing by reference

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

```
1  void f(int &n) {
2    n = /* some expression */ ;
3  };
4  int main() {
5    int i;
6    f(i);
7    // i now has the value that was set in the function
8  }
```

Reference syntax is cleaner than C 'pass by reference'

61. Results other than through return

Also good design:

- Return no function result,
- or return return status (0 is success, nonzero various informative statuses), and
- return other information by changing the parameters.
- *pass by reference*
- Parameters are sometimes classified 'input', 'output', 'throughput'.

62. Pass by reference example 1

Code:

```
1 // basic/setbyref.cpp
2 void f( int &i ) {
3   i = 5;
4 }
5 int main() {
6
7   int var = 0;
8   f(var);
9   cout << var << '\n';
```

Output:

```
5
```

Compare the difference with leaving out the reference.

63. Pass by reference example 2

```
bool can_read_value( int &value ) {
  // this uses functions defined elsewhere
  int file_status = try_open_file();
  if (file_status==0)
    value = read_value_from_file();
  return file_status==0;
}

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

Exercise 25

Write a `void` function *swap* of two parameters that exchanges the input values:

Code:

```
1 // func/swap.cpp
2 int i=1,j=2;
3 cout << i << "," << j << '\n';
4 swap(i,j);
5 cout << i << "," << j << '\n';
```

Output:

```
1,2
2,1
```

Exercise 26

Write a divisibility function that takes a number and a divisor, and gives:

- a `bool` return result indicating that the number is divisible, and
- a remainder as output parameter.

Code:

```
1 // func/divisible.cpp
2 cout << number;
3 if
      (is_divisible(number,divisor,remainder))
4   cout << " is divisible by ";
5 else
6   cout << " has remainder "
7        << remainder << " from ";
8 cout << divisor << '\n';
```

Output:

```
8 has remainder 2 from 3
8 is divisible by 4
```

Exercise 27

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:

```
1 // geom/rotate.cpp
2 const float pi = 2*acos(0.0);
3 float x{1.}, y{0.};
4 rotate(x,y,pi/4);
5 cout << "Rotated halfway: ("
6      << x << "," << y << ")" <<
       '\n';
7 rotate(x,y,pi/4);
8 cout << "Rotated to the y-axis: ("
9      << x << "," << y << ")" <<
       '\n';
```

Output:

```
Rotated halfway:
     (0.707107,0.707107)
Rotated to the y-axis:
     (0,1)
```

Recursion

Reference material

The following slides are a high-level introduction;
for details see: section Textbook, section 7.6

64. Recursion

A function is allowed to call itself, making it a recursive function.
For example, factorial:

$$5! = 5 \cdot 4 \cdot \cdots \cdot 1 = 5 \times 4!$$

You can define factorial as

$$F(n) = n \times F(n-1) \qquad \text{if } n > 1, \text{ otherwise } 1$$

```
int factorial( int n ) {
  if (n==1)
    return 1;
  else
    return n*factorial(n-1);
}
```

Exercise 28

The sum of squares:

$$S_n = \sum_{n=1}^{N} n^2$$

can be defined recursively as

$$S_1 = 1, \qquad S_n = n^2 + S_{n-1}.$$

Write a recursive function that implements this second definition. Test it on numbers that are input interactively.

Then write a program that prints the first 100 sums of squares.

How many squares do you need to sum before you get overflow? Can you estimate this number without running your program?

Exercise 29

It is possible to define multiplication as repeated addition:

Code:

```
1 // func/mult.cpp
2 int times( int number,int mult ) {
3   cout << "(" << mult << ")";
4   if (mult==1)
5     return number;
6   else
7     return number +
       times(number,mult-1);
8 }
```

Output:

```
Enter number and
     multiplier
recursive multiplication
 of 7 and 5:
     (5)(4)(3)(2)(1)35
```

Extend this idea to define powers as repeated multiplication.

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

Exercise 30

The Egyptian multiplication algorithm is almost 4000 years old. The result of multiplying $x \times n$ is:

if n is even:
  twice the multiplication $x \times (n/2)$;
otherwise if $n == 1$:
  x
otherwise:
  x plus the multiplication $x \times (n - 1)$

Extend the code of exercise 29 to implement this.

Food for thought: discuss the computational aspects of this algorithm to the traditional one of repeated addition.

Exercise 31

Write a recursive function for computing Fibonacci numbers:

$$F_0 = 1, \qquad F_1 = 1, \qquad F_n = F_{n-1} + F_{n-2}$$

First write a program that computes F_n for a value n that is input interactively.

Then write a program that prints out a sequence of Fibonacci numbers; set interactively how many.

More about functions

65. Default arguments

Functions can have default argument(s):

```
double distance( double x, double y=0. ) {
  return sqrt( (x-y)*(x-y) );
}
  ...
  d = distance(x); // distance to origin
  d = distance(x,y); // distance between two points
```

Any default argument(s) should come last in the parameter list.

66. Polymorphic functions

You can have multiple functions with the same name:

```
double average(double a,double b) {
  return (a+b)/2; }
double average(double a,double b,double c) {
  return (a+b+c)/3; }
```

Distinguished by type or number of input arguments: can not differ only in return type.

```
int f(int x);
string f(int x); // DOES NOT WORK
```

67. Useful idiom

Don't trace a function unless I say so:

```
void dosomething(double x,bool trace=false) {
  if (trace) // report on stuff
};
int main() {
  dosomething(1); // this one I trust
  dosomething(2); // this one I trust
  dosomething(3,true); // this one I want to trace!
  dosomething(4); // this one I trust
  dosomething(5); // this one I trust
```

Scope

Reference material

The following slides are a high-level introduction;
for details see: chapter Textbook, section 8

68. Lexical scope

Visibility of variables

```
int main() {
  int i;
  if ( something ) {
    int j;
    // code with i and j
  }
  int k;
  // code with i and k
}
```

69. Shadowing

```
int main() {
  int i = 3;
  if ( something ) {
    int i = 5;
  }
  cout << i << endl; // gives 3
  if ( something ) {
    float i = 1.2;
  }
  cout << i << endl; // again 3
}
```

Variable `i` is shadowed: invisible for a while.
After the lifetime of the shadowing variable, its value is unchanged from before.

Exercise 32

What is the output of this code?

```
// basic/shadowfalse.cpp
bool something{false};
int i = 3;
if ( something ) {
  int i = 5;
  cout << "Local: " << i << '\n';
}
cout << "Global: " << i << '\n';
if ( something ) {
  float i = 1.2;
  cout << i << '\n';
  cout << "Local again: " << i << '\n';
}
cout << "Global again: " << i << '\n';
```

70. Life time vs reachability

Even without shadowing, a variable can exist but be unreachable.

```
void f() {
  ...
}
int main() {
  int i;
  f();
  cout << i;
}
```

Arrays

71. Short vectors

Short vectors can be created by enumerating their elements:

```
1  // array/shortvector.cpp
2  #include <vector>
3  using std::vector;
4
5  int main() {
6    vector<int> evens{0,2,4,6,8};
7    vector<float> halves = {0.5, 1.5, 2.5};
8    auto halfloats = {0.5f, 1.5f, 2.5f};
9    cout << evens.at(0) << '\n';
10   return 0;
11 }
```

Exercise 33

1. Take the above snippet, compile, run.
2. Add a statement that alters the value of a vector element. Check that it does what you think it does.
3. Add a vector of the same length as the `evens` vector, containing odd numbers which are the even values plus 1?

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

72. Range over elements

A range-based for loop gives you directly the element values:

```
vector<float> my_data(N);
/* set the elements somehow */;
for ( float e : my_data )
  // statement about element e
```

Here there are no indices because you don't need them.

73. Range over elements, version 2

Same with `auto` instead of an explicit type for the elements:

```
for ( auto e : my_data )
  // same, with type deduced by compiler
```

74. Range over elements

Finding the maximum element

Code:

```
1 // array/dynamicmax.cpp
2 vector<int> numbers = {1,4,2,6,5};
3 int tmp_max = -2000000000;
4 for (auto v : numbers)
5   if (v>tmp_max)
6     tmp_max = v;
7 cout << "Max: " << tmp_max
8      << " (should be 6)" << '\n';
```

Output:

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

Exercise 34

Find the element with maximum absolute value in a vector. Use:

```
vector<int> numbers = {1,-4,2,-6,5};
```

Hint:

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

Exercise 35

Indicate for each of the following vector operations whether you prefer to use an indexed loop or a range-based loop. Give a short motivation.

- Count how many elements of a vector are zero.
- Find the location of the last zero.

75. Range over vector denotation

Code:

```
1 // array/rangedenote.cpp
2 for ( auto i : {2,3,5,7,9} )
3   cout << i << ",";
4 cout << '\n';
```

Output:

```
2,3,5,7,9,
```

76. Vector definition

Definition and/or initialization:

```
1 #include <vector>
2 using std::vector;
3
4 vector<type> name;
5 vector<type> name(size);
6 vector<type> name(size,init_value);
```

where

- `vector` is a keyword,
- `type` (in angle brackets) is any elementary type or class name,
- `name` of the vector is up to you, and
- `size` is the (initial size of the vector). This is an integer, or more precisely, a `size_t` parameter.
- Initialize all elements to `init_value`.
- If no default given, zero is used for numeric types.

77. Accessing vector elements

Square bracket notation (zero-based):

Code:

```
1 // array/assign.cpp
2 vector<int> numbers = {1,4};
3 numbers[0] += 3;
4 numbers[1] = 8;
5 cout << numbers[0] << ","
6      << numbers[1] << '\n';
```

Output:

```
4,8
```

With bound checking:

Code:

```
1 // array/assign.cpp
2 vector<int> numbers = {1,4};
3 numbers.at(0) += 3;
4 numbers.at(1) = 8;
5 cout << numbers.at(0) << ","
6      << numbers.at(1) << '\n';
```

Output:

```
4,8
```

Safer, slower.

78. Vector elements out of bounds

Square bracket notation:

Code:

```
1 // array/assignoutofbound.cpp
2 vector<int> numbers = {1,4};
3 numbers[-1] += 3;
4 numbers[2] = 8;
5 cout << numbers[0] << ","
6      << numbers[1] << '\n';
```

Output:

```
1,4
```

With bound checking:

Code:

```
1 // array/assignoutofbound.cpp
2 vector<int> numbers = {1,4};
3 numbers.at(-1) += 3;
4 numbers.at(2) = 8;
5 cout << numbers.at(0) << ","
6      << numbers.at(1) << '\n';
```

Output:

```
libc++abi: terminating
     with uncaught
     exception of type
     std::out_of_range:
     vector
```

Safer, slower.

79. Range over elements by reference

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

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

Code:

```
1 // array/vectorrangeref.cpp
2 vector<float> myvector
3   = {1.1, 2.2, 3.3};
4 for ( auto &e : myvector )
5   e *= 2;
6 cout << myvector.at(2) << '\n';
```

Output:

```
6.6
```

(Can also use `const auto& e` to prevent copying, but also prevent altering data.)

80. Example: multiplying elements

Example: multiply all elements by two:

Code:

```
1 // array/vectorrangeref.cpp
2 vector<float> myvector
3   = {1.1, 2.2, 3.3};
4 for ( auto &e : myvector )
5   e *= 2;
6 cout << myvector.at(2) << '\n';
```

Output:

```
6.6
```

81. Indexing the elements

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

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

Example: find the maximum element in the vector, and where it occurs.

Code:

```
1 // array/vectoridxmax.cpp
2 int tmp_idx = 0;
3 int tmp_max = numbers.at(tmp_idx);
4 for (int i=0; i<numbers.size(); ++i) {
5   int v = numbers.at(i);
6   if (v>tmp_max) {
7     tmp_max = v; tmp_idx = i;
8   }
9 }
10 cout << "Max: " << tmp_max
11      << " at index: " << tmp_idx << '\n';
```

Output:

```
Max: 6.6 at
     index: 3
```

82. A philosophical point

Conceptually, a `vector` can correspond to a set of things, and the fact that they are indexed is purely incidental, or it can correspond to an ordered set, and the index is essential. If your algorithm requires you to access all elements, it is important to think about which of these cases apply, since there are two different mechanism.

Exercise 36

Find the location of the first negative element in a vector.

Which mechanism do you use?

Exercise 37

Create a `vector` x of `float` elements, and set them to random values. (Use the C random number generator for now.)

Now normalize the vector in L_2 norm and check the correctness of your calculation, that is,

1. Compute the L_2 norm of the vector:

$$\|v\| \equiv \sqrt{\sum_i v_i^2}$$

2. Divide each element by that norm;
3. The norm of the scaled vector should now by 1. Check this.
4. Bonus: your program may be printing 1, but is it actually 1? Investigate.

What type of loop are you using?

83. Indexing with pre/post increment

Indexing in `while` loop and such:

```
x = a.at(i++); /* is */ x = a.at(i); i++;
y = b.at(++i); /* is */ i++; y = b.at(i);
```

84. Vector copy

Vectors can be copied just like other datatypes:

Code:

```
1 // array/vectorcopy.cpp
2 vector<float> v(5,0), vcopy;
3 v.at(2) = 3.5;
4 vcopy = v;
5 vcopy.at(2) *= 2;
6 cout << v.at(2) << ","
7      << vcopy.at(2) << '\n';
```

Output:

```
3.5,7
```

85. Vector methods

A vector is an object, with methods.

Given `vector<sometype> x`:

- Get elements, including bound checking, with `ar.at(3)`. Note: (zero-based indexing).
- (also get elements with `ar[3]`: see later discussion.)
- Size: `ar.size()`.
- Other functions: `front`, `back`, `empty`.
- With iterators (see later): `insert`, `erase`

86. Your first encounter with templates

vector is a 'templated class': *vector*<*X*> is a vector-of-*X*.

Code behaves as if there is a class definition for each type:

```
class vector<int> {
public:
  size(); at(); // stuff
}
```

```
class vector<float> {
public:
  size(); at(); // stuff
}
```

Actual mechanism uses templating: the type is a parameter to the class definition.

Dynamic behaviour

87. Dynamic vector extension

Extend a vector's size with `push_back`:

Code:

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

Output:

```
6
35
```

Similar functions: `pop_back`, `insert`, `erase`.
Flexibility comes with a price.

88. When to push back and when not

Known vector size:

```
int n = get_inputsize();
vector<float> data(n);
for ( int i=0; i<n; i++ ) {
  auto x = get_item(i);
  data.at(i) = x;
}
```

Unknown vector size:

```
vector<float> data;
float x;
while ( next_item(x) ) {
  data.push_back(x);
}
```

If you have a guess as to size: `data.reserve(n)`.

(Issue with array-of-object: in left code, constructors are called twice.)

89. Filling in vector elements

You can push elements into a vector:

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

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

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

90. Filling in vector elements

With subscript:

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

You can also use `new` to allocate*:

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

*Considered bad practice. Do not use.

91. Timing the ways of filling a vector

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

Vectors and functions

92. Vector as function return

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

Example: this function creates a vector, with the first element set to the size:

Code:

```
1  // array/vectorreturn.cpp
2  vector<int> make_vector(int n) {
3    vector<int> x(n);
4    x.at(0) = n;
5    return x;
6  }
7      /* ... */
8    vector<int> x1 = make_vector(10);
9    // "auto" also possible!
10   cout << "x1 size: " << x1.size()
        << '\n';
11   cout << "zero element check: " <<
        x1.at(0) << '\n';
```

Output:

```
x1 size: 10
zero element check: 10
```

93. Vector as function argument

You can pass a vector to a function:

```
double slope( vector<double> v ) {
  return v.at(1)/v.at(0);
};
```

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

94. Vector pass by value example

Code:

```
1 // array/vectorpassnot.cpp
2 void set0
3   ( vector<float> v,float x )
4 {
5   v.at(0) = x;
6 }
7     /* ... */
8   vector<float> v(1);
9   v.at(0) = 3.5;
10  set0(v,4.6);
11  cout << v.at(0) << '\n';
```

Output:

```
3.5
```

- Vector is copied
- 'Original' in the calling environment not affected
- Cost of copying?

95. Vector pass by reference

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

Code:

```
1 // array/vectorpassref.cpp
2 void set0
3   ( vector<float> &v,float x )
4 {
5   v.at(0) = x;
6 }
7     /* ... */
8   vector<float> v(1);
9   v.at(0) = 3.5;
10  set0(v,4.6);
11  cout << v.at(0) << '\n';
```

Output:

```
4.6
```

- Parameter vector becomes alias to vector in calling environment
  $\Rightarrow$ argument *can* be affected.
- No copying cost
- What if you want to avoid copying cost, but need not alter the argument?

96. Vector pass by const reference

Passing a vector that does not need to be altered:

```
int f( const vector<int> &ivec ) { ... }
```

- Zero copying cost
- Not alterable, so: safe!
- (No need for pointers!)

Exercise 38

Revisit exercise 37 and introduce a function for computing the L_2 norm.

(hints for the next exercise)

Random numbers:

```
// high up in your code:
#include <random>
using std::rand;

// in your main or function:
float r = 1.*rand()/RAND_MAX;
// gives random between 0 and 1
```

(You will learn a better random later)

Exercise 39

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

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

This creates a vector of random values of a specified length, and then makes a sorted copy of it.

Instead of making a sorted copy, sort in-place (overwrite original data with sorted data):

```
int length = 10;
vector<float> values = random_vector(length);
sort(values); // the vector is now sorted
```

Find arguments for/against that approach.

(Note: C++ has sorting functions built in.)

Vectors in classes

97. Can you make a class around a vector?

You may want a class of objects that contain a vector. For instance, you may want to name your vectors.

```
1 class named_field {
2 private:
3   vector<double> values;
4   string name;
```

The problem here is when and how that vector is going to be created.

98. Create the contained vector

Use initializers for creating the contained vector:

```
1  class named_field {
2  private:
3    string name;
4    vector<double> values;
5  public:
6    named_field( string name,int n )
7      : name(name),
8        values(vector<double>(n)) {
9    };
10 };
```

Even shorter:

```
1    named_field( string name,int n )
2      : name(name),values(n) {
3    };
```

Multi-dimensional arrays

99. Multi-dimensional 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.

100. Matrix class

```
1  // array/matrix.cpp
2  class matrix {
3  private:
4    vector<vector<double>> elements;
5  public:
6    matrix(int m,int n)
7      : elements(
8                  vector<vector<double>>(m,vector<double>(n))
9                  ) {
10   }
11   void set(int i,int j,double v) {
12     elements.at(i).at(j) = v;
13   };
14   double get(int i,int j) {
15     return elements.at(i).at(j);
16   };
```

(Can you combine the *get/set* methods, using **??**?)

Exercise 40

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

Exercise 41

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 // array/matrix.cpp
2 A.set(3.);
3 cout << "Sum of elements: "
4      << A.totalsum() << '\n';
```

Output:

```
Sum of elements: 30
```

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

101. Matrix class; better design

Better idea:

```
// array/matrixclass.cpp
class matrix {
private:
  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 );
  };
    /* ... */
};
```

(Old-style solution: use cpp macro)

Exercise 42

In the matrix class of the previous slide, why are `m,n` stored explicitly, unlike in the matrix class of section **??**?

Exercise 43

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

Implement matrix-matrix multiplication.

102. Pascal's triangle

Pascal's triangle contains binomial coefficients:

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

where

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

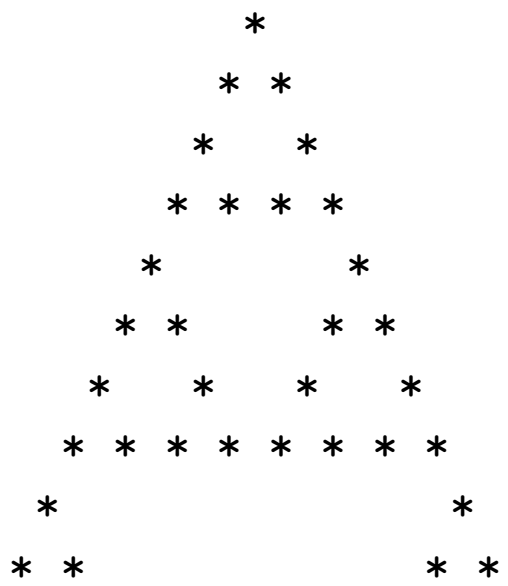
The coefficients can be computed from the recurrence

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

(There are other formulas. Why are they less preferable?)

Exercise 44

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



103. Exercise continued

- The object needs to have an array internally. The easiest solution is to make an array of size $n \times n$.
- Your program should accept:
  1. an integer for the size
  2. any number of integers for the modulo; if this is zero, stop, otherwise print stars as described above.

Optional exercise 45

Extend the Pascal exercise:
Optimize your code to use precisely enough space for the coefficients.

Turn it in!

- Write a program that accepts:
  1. one integer: the height of the triangle. You should use this to construct a `PascalTriangle` object that contains the binomial coefficients. Then:
  2. a number of modulos with which to print the triangle. A value of zero indicates that your program should stop.

  The tester will search for stars in your output and test that you have the right number in each line.
- If you have compiled your program, do a test run:

  ```
  coe_pascal yourprogram.cc
  ```
- Is it Submit if it is correct:

  ```
  coe_pascal -s yourprogram.cc
  ```
- If you don't manage to get your code working correctly, you can submit as incomplete with

  ```
  coe_pascal -i yourprogram.cc
  ```

Other array stuff

104. Array class

Static arrays:

```
#include <array>
std::array<int,5> fiveints;
```

- Size known at compile time.
- Vector methods that do not affect storage
- Zero overhead.

105. Random walk exercise

```
// rand/walk_lib_vec.cpp
class Mosquito {
private:
  vector<float> pos;
public:
  Mosquito( int d )
    : pos( vector<float>(d,0.f) ) { };

// rand/walk_lib_vec.cpp
void step() {
  int d = pos.size();
  auto incr = random_step(d);
  for (int id=0; id<d; ++id)
    pos.at(id) += incr.at(id);
};
```

Finish the implementation. Do you get improvement from using the array class?

106. Span

Create a `span` from a `vector`:

```
#include <span>
vector<double> v;
auto v_span = std::span<double>( v.data(),v.size() );
```

Array creation

C-style arrays still exist,

```
// array/staticinit.cpp
{
  int numbers[] = {5,4,3,2,1};
  cout << numbers[3] << '\n';
}
{
  int numbers[5]{5,4,3,2,1};
  numbers[3] = 21;
  cout << numbers[3] << '\n';
}
```

but you shouldn't use them.
Prefer to use *array* class (not in this course)
or span (C++20; very advanced)

I/O

107. I/O, what's it about?

Input: getting data from keyboard or file into your program.

Output: getting data from your program to screen or file.

The fmtlib library

108. Simple example

The basic usage is:

```
int i=2;
format("string {} brace expressions",i);
```

Format string, and arguments.

109. Displaying the format result

Use `cout` or (C++23) *print*:

Code:

```
1 // io/fmtbasic.cpp
2 cout << format("{}\n",2);
3 string hello_string = format
4   ("{} {}!","Hello","world");
5 cout << hello_string << '\n';
6 print
7   ("{0}, {0}
      {1}!\n","Hello","world");
```

Output:

```
2
Hello world!
Hello, Hello world!
```

110. Right align

Right-align with > character and width:

Code:

```
1 // io/fmtlib.cpp
2 for (int i=10; i<2000000000; i*=10)
3   fmt::print("{:>6}\n",i);
```

Output:

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

111. Padding character

Other than space for padding:

Code:

```
1 // io/fmtlib.cpp
2 for (int i=10; i<2000000000; i*=10)
3   fmt::print("{0:.>6}\n",i);
```

Output:

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

112. Number bases

Code:

```
1 // io/fmtlib.cpp
2 fmt::print
3   ("{0} = {0:b} bin\n",17);
4 fmt::print
5   ("     = {0:o} oct\n",17);
6 fmt::print
7   ("     = {0:x} hex\n",17);
```

Output:

```
17 = 10001 bin
     = 21 oct
     = 11 hex
```

113. Hex numbers

Display the numbers 0 . . . 255 in a square

```
for (int i=0; i<16; i++)
for (int j=0; j<16; j++)
// output 16*i+j on base 16
```

```
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 46

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
```

114. Float and fixed

Floating point or normalized exponential with *e* specifier
fixed: use decimal point if it fits, *m.n* specification

Code:

```
1 // io/fmtfloat.cpp
2 x = 1.234567;
3 for (int i=0; i<6; ++i) {
4   fmt::print
5     ("{0:.3e}/{0:7.4}\n",x);
6   x *= 10;
7 }
```

Output:

```
1.235e+00/  1.235
1.235e+01/  12.35
1.235e+02/  123.5
1.235e+03/   1235
1.235e+04/1.235e+04
1.235e+05/1.235e+05
```

115. Treatment of leading sign

Positive sign always, nothing, blank:

Code:

```
1 // io/fmtsci.cpp
2 float pi=3.14159f;
3 fmt::print("|{:+.2e}|{:+.2e}|\n",
4             pi,-pi);
5 fmt::print("|{:-.2e}|{:-.2e}|\n",
6             pi,-pi);
7 fmt::print("|{: .2e}|{: .2e}|\n",
8             pi,-pi);
```

Output:

```
|+3.14e+00|-3.14e+00|
|3.14e+00|-3.14e+00|
| 3.14e+00|-3.14e+00|
```

116. fmtlib: usage

```
#include <fmt/format.h>
using fmt::format;
```

117. fmtlib: installing

- Download: `https://github.com/fmtlib/fmt`
- Cmake installation
- add *lib/pkgconfig* to *PKG_CONFIG_PATH*

118. fmtlib: compilation

Compilation on the commandline:

```
g++ -o myprog myprog.cpp \
    $( pkg-config --cflags fmt ) \
    $( pkg-config --libs fmt )
```

119. fmtlib: compilation'

Using CMake:

```
find_package( PkgConfig REQUIRED )
pkg_check_modules( FMTLIB REQUIRED fmt )
target_include_directories(
        ${PROGRAM_NAME} PUBLIC ${FMTLIB_INCLUDE_DIRS} )
target_link_directories(
        ${PROGRAM_NAME} PUBLIC ${FMTLIB_LIBRARY_DIRS} )
target_link_libraries(
        ${PROGRAM_NAME} PUBLIC ${FMTLIB_LIBRARIES} )
set_target_properties(
        ${PROGRAM_NAME} PROPERTIES
        BUILD_RPATH   "${FMTLIB_LIBRARY_DIRS}"
        INSTALL_RPATH "${FMTLIB_LIBRARY_DIRS}"
    )
```

120. fmtlib: use through pkg-config

When you install fmtlib, note the location of the `.pc` file, then

```
export
    PKG_CONFIG_PATH=/the/location/from/fmtlib:${PKG_CONFIG_PATH}
```

in your `.bashrc` (Mac users: `.zshrc`)

Formatted stream output

121. Formatted output

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

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

122. Default unformatted output

Code:

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

Output:

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

123. Reserve space

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

Code:

```
1  // io/width.cpp
2  #include <iomanip>
3  using std::setw;
4      /* ... */
5    cout << "Width is 6:" << '\n';
6    for (int i=1; i<200000000; i*=10)
7      cout << "Number: "
8           << setw(6) << i << '\n';
9    cout << '\n';
10
11   // `setw' applies only once:
12   cout << "Width is 6:" << '\n';
13   cout << ">"
14        << setw(6) << 1 << 2 << 3 <<
        '\n';
15   cout << '\n';
```

Output:

```
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
```

124. Padding character

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

Code:

```
1 // io/formatpad.cpp
2 #include <iomanip>
3 using std::setfill;
4 using std::setw;
5     /* ... */
6   for (int i=1; i<200000000; i*=10)
7     cout << "Number: "
8          << setfill('.')
9          << setw(6) << i
10         << '\n';
```

Output:

```
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.

125. Left alignment

Instead of right alignment you can do left:

Code:

```
1 // io/formatleft.cpp
2 #include <iomanip>
3 using std::left;
4 using std::setfill;
5 using std::setw;
6     /* ... */
7   for (int i=1; i<200000000; i*=10)
8     cout << "Number: "
9          << left << setfill('.')
10         << setw(6) << i << '\n';
```

Output:

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

126. Number base

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

Code:

```
 1 // io/format16.cpp
 2 #include <iomanip>
 3 using std::setbase;
 4 using std::setfill;
 5     /* ... */
 6   cout << setbase(16)
 7        << setfill(' ');
 8   for (int i=0; i<16; ++i) {
 9     for (int j=0; j<16; ++j)
10       cout << i*16+j << " " ;
11     cout << '\n';
12   }
```

Output:

```
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
```

Exercise 47

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

Code:

```
1 // io/quasifix.cpp
2 string quasifix(double);
3 int main() {
4   for ( auto x : { 1.5, 12.32,
       123.456, 1234.5678 } )
5     cout << quasifix(x) << '\n';
```

Output:

```
   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.

127. Hexadecimal

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

Code:

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

Output:

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

Back to decimal:

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

Floating point formatting

128. Floating point precision

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

Code:

```
1  // io/formatfloat.cpp
2  #include <iomanip>
3  using std::left;
4  using std::setfill;
5  using std::setw;
6  using std::setprecision;
7      /* ... */
8    x = 1.234567;
9    for (int i=0; i<10; ++i) {
10     cout << setprecision(4) << x <<
        '\n';
11     x *= 10;
12   }
```

Output:

```
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.

129. Fixed point precision

Fixed precision applies to fractional part:

Code:

```
1 // io/fix.cpp
2 x = 1.234567;
3 cout << fixed;
4 for (int i=0; i<10; ++i) {
5   cout << setprecision(4) << x <<
      '\n';
6   x *= 10;
7 }
```

Output:

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

(Notice the rounding)

130. Aligned fixed point output

Combine width and precision:

Code:

```
1 // io/align.cpp
2   x = 1.234567;
3   cout << fixed;
4   for (int i=0; i<10; ++i) {
5     cout << setw(10) <<
       setprecision(4) << x
6          << '\n';
7     x *= 10;
8   }
```

Output:

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

131. Scientific notation

Combining width and precision:

Code:

```
1 // io/iof.cpp
2 x = 1.234567;
3 cout << scientific;
4 for (int i=0; i<10; ++i) {
5   cout << setw(10) <<
       setprecision(4)
6        << x << '\n';
7   x *= 10;
8 }
9 cout << '\n';
```

Output:

```
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
```

File output

132. Text output to file

Use:

Code:

```
1 // io/fio.cpp
2 #include <fstream>
3 using std::ofstream;
4     /* ... */
5   ofstream file_out;
6   file_out.open
7     ("fio_example.out");
8     /* ... */
9   file_out << number << '\n';
10  file_out.close();
```

Output:

```
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'.

133. Binary I/O

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

Code:

```
1 // io/fiobin.cpp
2 cout << "Writing: " << x << '\n';
3 ofstream file_out;
4 file_out.open
5   ("fio_binary.out",ios::binary);
6 file_out.write
7   (reinterpret_cast<char*>(&x),
8    sizeof(double));
9 file_out.close();
```

Output:

```
Writing: 0.841471
```

write takes an address and the number of bytes.

134. Binary I/O'

Input is mirror of the output:

Code:

```
1 // io/fiobin.cpp
2 ifstream file_in;
3 file_in.open
4   ("fio_binary.out",ios::binary);
5 file_in.read
6   (reinterpret_cast<char*>(&x),
7    sizeof(double));
8 file_in.close();
9 cout << "Read   : " << x << '\n';
```

Output:

```
Read   : 0.841471
```

Cout on classes (for future reference)

135. Redefine less-less

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

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

Strings

Characters

136. Characters and ints

- Type `char`;
- represents '7-bit ASCII': printable and (some) unprintable characters.
- Single quotes: `char c = 'a'`

137. Char / int equivalence

Equivalent to (short) integer:

Code:

```
1 // string/intchar.cpp
2 char ex = 'x';
3 int x_num = ex, y_num = ex+1;
4 char why = y_num;
5 cout << "x is at position " << x_num
6      << '\n';
7 cout << "one further lies " << why
8      << '\n';
```

Output:

```
x is at position 120
one further lies y
```

Also: `'x'-'a'` is distance a--x

Exercise 48

Write a program that accepts an integer $1 \cdots 26$ and prints the so-manieth letter of the alphabet.

Extend your program so that if the input is negative, it prints the minus-so-manieth uppercase letter of the alphabet.

Strings

138. String declaration

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

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

(Do not use the C legacy mechanisms.)

139. String creation

A string variable contains a string of characters.

```
string txt;
```

You can initialize the string variable or assign it dynamically:

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

140. Quotes in strings

You can escape a quote, or indicate that the whole string is to be taken literally:

Code:

```
1 // string/quote.cpp
2 string
3   one("a b c"),
4   two("a \"b\" c"),
5   three( R"("a ""b """c)" );
6 cout << one << '\n';
7 cout << two << '\n';
8 cout << three << '\n';
```

Output:

```
a b c
a "b" c
"a ""b """c
```

141. Concatenation

Strings can be *concatenated*:

Code:

```
1 // string/strings.cpp
2 string my_string, space{" "};
3 my_string = "foo";
4 my_string += space + "bar";
5 cout << my_string << ": " <<
       my_string.size() << '\n';
```

Output:

```
foo bar: 7
```

142. String indexing

You can query the *size*:

Code:

```
1 // string/strings.cpp
2 string five_text{"fiver"};
3 cout << five_text.size() << '\n';
```

Output:

```
5
```

or use subscripts:

Code:

```
1 // string/stringsub.cpp
2 string digits{"0123456789"};
3 cout << "char three: "
4      << digits[2] << '\n';
5 cout << "char four : "
6      << digits.at(3) << '\n';
```

Output:

```
char three: 2
char four : 3
```

143. Ranging over a string

Same as ranging over vectors.

Range-based for:

Code:

```
1 // string/stringrange.cpp
2 cout << "By character: ";
3 for ( char c : abc )
4   cout << c << " ";
5 cout << '\n';
```

Output:

```
By character: a b c
```

Ranging by index:

Code:

```
1 // string/stringrange.cpp
2 string abc = "abc";
3 cout << "By character: ";
4 for (int ic=0; ic<abc.size(); ic++)
5   cout << abc[ic] << " ";
6 cout << '\n';
```

Output:

```
By character: a b c
```

144. Range with reference

Range-based for makes a copy of the element
You can also get a reference:

Code:

```
1 // string/stringrange.cpp
2 for ( char &c : abc )
3   c += 1;
4 cout << "Shifted: " << abc << '\n';
```

Output:

```
Shifted: bcd
```

Review quiz 13

True or false?

1. `'0'` is a valid value for a `char` variable
   `/poll "single-quote 0 is a valid char" "T" "F"`
2. `"0"` is a valid value for a `char` variable
   `/poll "double-quote 0 is a valid char" "T" "F"`
3. `"0"` is a valid value for a `string` variable
   `/poll "double-quote 0 is a valid string" "T" "F"`
4. `'a'+'b'` is a valid value for a `char` variable
   `/poll "adding single-quote chars is a valid char" "T" "F"`

Exercise 49

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

$$s \equiv 3\text{: "acdz"} \Rightarrow \text{"dfgc"}.$$

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

145. More vector methods

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

Code:

```
1 // string/strings.cpp
2 string five_chars;
3 cout << five_chars.size() << '\n';
4 for (int i=0; i<5; ++i)
5   five_chars.push_back(' ');
6 cout << five_chars.size() << '\n';
```

Output:

```
0
5
```

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

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

Exercise 50

Write a function to print out the digits of a number: `156` should print `one five six`. You need to convert a digit to a string first; can you think of more than one way to do that?

Start by writing a program that reads a single digit and prints its name.

For the full program it is easiest to generate the digits last-to-first. Then figure out how to print them reversed.

Optional exercise 51

Write a function to convert an integer to a string: the input 215 should give `two hundred fifteen`, et cetera.

146. String stream

Like cout (including conversion from quantity to string), but to object, not to screen.

- Use the << operator to build it up; then
- use the *str* method to extract the string.

```
1 #include <sstream>
2 stringstream s;
3 s << "text" << 1.5;
4 cout << s.str() << endl;
```

147. String an object, 1

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

```
1 // geom/pointfunc.cpp
2   string as_string() {
3     stringstream ss;
4     ss << "(" << x << "," << y << ")";
5     return ss.str();
6   };
7     /* ... */
8 std::ostream& operator<<
9 (std::ostream &out,Point &p) {
10   out << p.as_string(); return out;
11 };
```

148. String an object, 2

Redefine the less-less operator to use this.

```
1 // geom/pointfunc.cpp
2 Point p1(1.,2.);
3 cout << "p1 " << p1
4      << " has length "
5      << p1.length() << "\n";
```

Exercise 52

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

Code:

```
1 // io/quasifix.cpp
2 string quasifix(double);
3 int main() {
4   for ( auto x : { 1.5, 12.32,
       123.456, 1234.5678 } )
5     cout << quasifix(x) << '\n';
```

Output:

```
   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.

Objects

Classes

149. Definition of object/class

An object is an entity that you can request to do certain things. These actions are the *methods*, and to make these possible the object probably stores data, the *members*.

When designing a class, first ask yourself:
'what functionality should the objects support'.

A class is a user-defined type; an object is an instance of that type.

150. Running example

We are going to build classes for points/lines/shapes in the plane.

```
1 class Point {
2     /* stuff */
3 };
4 int main () {
5   Point p; /* stuff */
6 }
```

Exercise 53

Thought exercise: what are some of the actions that a point object should be capable of?

151. Object functionality

Small illustration: point objects.

Code:

```
1 // object/functionality.cpp
2 Point p(1.,2.);
3 cout << "distance to origin "
4      << p.distance_to_origin()
5      << '\n';
6 p.scaleby(2.);
7 cout << "distance to origin "
8      << p.distance_to_origin()
9      << '\n'
10     << "and angle " << p.angle()
11     << '\n';
```

Output:

```
distance to origin
     2.23607
distance to origin
     4.47214
and angle 1.10715
```

Note the 'dot' notation.

Exercise 54

Thought exercise:
What data does the object need to store to be able to calculate angle and distance to the origin?
Is there more than one possibility?

152. The object workflow

- First define the class, with data and function members:

```
class MyObject {
  // define class members
  // define class methods
};
```

  (details later) typically before the *main*.

- You create specific objects with a declaration

```
MyObject
  object1( /* .. */ ),
  object2( /* .. */ );
```

- You let the objects do things:

```
object1.do_this();
x = object2.do_that( /* ... */ );
```

153. Construct an object

The declaration of an object `x` of class `Point`; the coordinates of the point are initially set to `1.5,2.5`.

```
Point x(1.5, 2.5);
```

```
1  class Point {
2  private: // data members
3    double x,y;
4  public: // function members
5    Point
6      (double x_in,double y_in){
7        x = x_in; y = y_in;
8    };
9    /* ... */
10 };
```

Use the constructor to create an object of a class:
function with same name as the class.
(but no return type!)

154. Private and public

Best practice we will use:

```
class MyClass {
private:
  // data members
public:
  // methods
}
```

- Data is private: not visible outside of the objects.
- Methods are public: can be used in the code that uses objects.
- You can have multiple private/public sections, in any order.

Methods

155. Class methods

Definition and use of the *distance* function:

Code:

```
1 // geom/pointclass.cpp
2 class Point {
3 private:
4   float x,y;
5 public:
6   Point(float in_x,float in_y) {
7     x = in_x; y = in_y; };
8   float distance_to_origin() {
9     return sqrt( x*x + y*y );
10  };
11 };
12    /* ... */
13  Point p1(1.0,1.0);
14  float d = p1.distance_to_origin();
15  cout << "Distance to origin: "
16       << d << '\n';
```

Output:

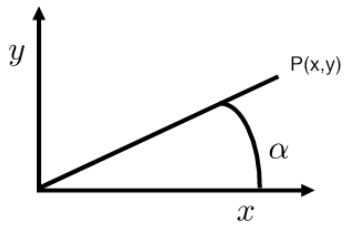
```
Distance to origin:
     1.41421
```

156. Class methods

- Methods look like ordinary functions,
- except that they can use the data members of the class, for instance `x,y`;
- Methods can only be used on an object with the 'dot' notation. They are not independently defined.

Exercise 55

Add a method *angle* to the *Point* class. How many parameters does it need?



Hint: use the function *atan* or *atan2*.

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

Exercise 56

Make a class *GridPoint* for points that have only integer coordinates. Implement a function *manhattan_distance* which gives the distance to the origin counting how many steps horizontal plus vertical it takes to reach that point.

157. Food for thought: constructor vs data

The arguments of the constructor imply nothing about what data members are stored!

Example: create a point in x,y Cartesian coordinates, but store $r,theta$ polar coordinates:

```
1 #include <cmath>
2 class Point {
3 private: // members
4   double r,theta;
5 public: // methods
6   Point( double x,double y ) {
7     r = sqrt(x*x+y*y);
8     theta = atan2(y/x);
9   }
```

Note: no change to outward API.

Exercise 57

Discuss the pros and cons of this design:

```
1  class Point {
2  private:
3    double x,y,r,theta;
4  public:
5    Point(double xx,double yy) {
6      x = xx; y = yy;
7      r = // sqrt something
8      theta = // something trig
9    };
10   double angle() { return alpha; };
11 };
```

158. Data access in methods

You can access data members of other objects of the same type:

```
1 class Point {
2 private:
3   double x,y;
4 public:
5  void flip() {
6     Point flipped;
7     flipped.x = y; flipped.y = x;
8     // more
9   };
10 };
```

(Normally, data members should not be accessed directly from outside an object)

Exercise 58

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.

Review quiz 14

T/F?

- A class is primarily determined by the data it stores.
  `/poll "Class determined by its data" "T" "F"`
- A class is primarily determined by its methods.
  `/poll "Class determined by its methods" "T" "F"`
- If you change the design of the class data, you need to change the constructor call.
  `/poll "Change data, change constructor proto too" "T" "F"`

159. Methods that alter the object

For instance, you may want to scale a vector by some amount:

Code:

```
1 // geom/pointscaleby.cpp
2 class Point {
3     /* ... */
4   void scaleby( double a ) {
5     x *= a; y *= a; };
6     /* ... */
7 };
8     /* ... */
9   Point p1(1.,2.);
10  cout << "p1 to origin "
11       << p1.length() << '\n';
12  p1.scaleby(2.);
13  cout << "p1 to origin "
14       << p1.length() << '\n';
```

Output:

```
p1 to origin 2.23607
p1 to origin 4.47214
```

Data initialization

160. Member default values

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

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

Each object will have its members initialized to these values.

161. Data initialization

The naive way:

```
1 class Point {
2 private:
3   double x,y;
4 public:
5   Point( double in_x,
6          double in_y ) {
7     x = in_x; y = in_y;
8   };
```

The preferred way:

```
1 // geom/pointinit.cpp
2 class Point {
3 private:
4   double x,y;
5 public:
6   Point( double in_x,
7          double in_y )
8     : x(in_x),y(in_y) {
9   }
```

Explanation later. It's technical.

Interaction between objects

162. Methods that create a new object

Code:

```
1 // geom/pointscale.cpp
2 class Point {
3     /* ... */
4   Point scale( double a ) {
5     auto scaledpoint =
6         Point( x*a, y*a );
7     return scaledpoint;
8   };
9     /* ... */
10  cout << "p1 to origin "
11       << p1.dist_to_origin()
12       << '\n';
13  Point p2 = p1.scale(2.);
14  cout << "p2 to origin "
15       << p2.dist_to_origin()
16       << '\n';
```

Output:

```
p1 to origin 2.23607
p2 to origin 4.47214
```

Note the 'anonymous *Point* object' in the *scale* method.

163. Anonymous objects

Create a point by scaling another point:

```
new_point = old_point.scale(2.81);
```

Two ways of handling the `return` statement:

Naive:

```
1 // geom/pointscale.cpp
2 Point Point::scale( double a )
      {
3   Point scaledpoint =
4     Point( x*a, y*a );
5   return scaledpoint;
6 };
```

Creates point, copies it to *new_point*

Better:

```
1 // geom/pointscale.cpp
2 Point Point::scale( double a )
      {
3   return Point( x*a, y*a );
4 };
```

Creates point, moves it directly to *new_point*

'move semantics' and 'copy elision':
compiler is pretty good at avoiding copies

Exercise 59

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?

164. Constructor/destructor

Constructor: function that gets called when you create an object.

```
MyClass {
public:
  MyClass( /* args */ ) { /* construction */ }
  /* more */
};
```

If you don't define it, you get a default.

Destructor (rarely used):
function that gets called when the object goes away, for instance when you leave a scope.

165. Using the default constructor

No constructor explicitly defined;

You recognize the default constructor in the main by the fact that an object is defined without any parameters.

Code:

```
1  // object/default.cpp
2  class IamOne {
3  private:
4    int i=1;
5  public:
6    void print() {
7      cout << i << '\n';
8    };
9  };
10     /* ... */
11   IamOne one;
12   one.print();
```

Output:

```
1
```

166. Default constructor

Refer to *Point* definition above.
Consider this code that looks like variable declaration, but for objects:

```
Point p1(1.5, 2.3);
Point p2;
p2 = p1.scaleby(3.1);
```

Compiling gives an error (g++; different for intel):

```
pointdefault.cpp: In function 'int main()':
pointdefault.cpp:32:21: error: no matching function for call to
                  'Point::Point()'
```

167. Default constructor

The problem is with *p2*:

```
Point p1(1.5, 2.3);
Point p2;
```

- *p1* is created with your explicitly given constructor;
- *p2* uses the default constructor:

  ```
  Point() {};
  ```

- default constructor is there by default, unless you define another constructor.
- you can redefine the default constructor:

  ```
  // geom/pointdefault.cpp
  Point() {};
  Point( double x,double y )
    : x(x),y(y) {};
  ```

  (but often you can avoid needing it)

Exercise 60

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;
```

168. Classes for abstract objects

Objects can model fairly abstract things:

Code:

```
1 // object/stream.cpp
2 class Stream {
3 private:
4   int last_result{0};
5 public:
6   int next() {
7     return last_result++; };
8 };
9
10 int main() {
11   Stream ints;
12   cout << "Next: "
13        << ints.next() << '\n';
14   cout << "Next: "
15        << ints.next() << '\n';
16   cout << "Next: "
17        << ints.next() << '\n';
```

Output:

```
Next: 0
Next: 1
Next: 2
```

169. Preliminary to the following exercise

A prime number generator has:
an API of just one function: `nextprime`

To support this it needs to store:
an integer `last_prime_found`

Programming Project Exercise 61

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:

```
// primes/6primesbyclass.cpp
cin >> nprimes;
primegenerator sequence;
while (sequence.number_of_primes_found()<nprimes) {
  int number = sequence.nextprime();
  cout << "Number " << number << " is prime" << '\n';
}
```

Programming Project Exercise 62

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.

First formulate the quantor structure of this statement, then translate that top-down to code, using the generator you developed above.

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.

170. A Goldbach corollary

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:

$$\forall_{r\,\text{prime}} \exists_{p,q\,\text{prime}} : r = (p+q)/2 \text{ is prime.}$$

Programming Project Exercise 63

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.

Use your prime generator. 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.

171. Hierarchical object relations

Hierarchical relations between classes:

- each object in class A is also in class B.
  but not conversely
- Example: each manager is an employee
- Example: each square is a rectangle

172. Example of class hierarchy

- Class *Employee*:

```
class Employee {
private:
  int number,salary;
/* ... */
};
```

- class *Manager* is subclass of *Employee*
  (every manager is an employee, with number and salary)
- Manager has extra field *n_minions*

How do we implement this?

173. Another example: multiple subclasses

- Example: both triangle and square are polygons.
- You can implement a method `draw` for both triangle/square
- ... or write it once for polygon, and then use that.

174. Terminology

Derived classes *inherit* data and methods from the base class: base class data and methods are accessible in objects of the derived class.

- Example: `Polygon` is the *base class*
  `Triangle` is a *derived class*
  Triangle has `corners` because Polygon has
- `Employee` is the *base class*.
  `Manager` is a *derived class*
  Manager has `employee_number` because Employee has

175. Base/Derived example

```
class Polygon {
protected:
vector<Point> corners;
public:
int ncorners() { return corners.size(); };
};
class Triangle : public Polygon {
/* constructor omitted */
};
int main () {
  Triangle t;
  cout << t.ncorners(); // prints 3, we hope
```

176. Examples for base and derived cases

General *FunctionInterpolator* class with method *value_at*. Derived classes:

- *LagranceInterpolator* with *add_point_and_value*;
- *HermiteInterpolator* with *add_point_and_derivative*;
- *SplineInterpolator* with *set_degree*.

177. General case, special case

You can have classes where an object of one class is a special case of the other class. You declare that as

```
1 class General {
2 protected: // note!
3  int g;
4 public:
5  void general_method() {};
6 };
7
8 class Special : public General {
9 public:
10   void special_method() { g = ... };
11 };
```

178. Inheritance: derived classes

Derived class `Special` *inherits* methods and data from base class `General`:

```
1 int main() {
2   Special special_object;
3   special_object.general_method();
4   special_object.special_method();
5 }
```

Members of the base class need to be `protected`, not `private`, to be inheritable.

179. Constructors

When you run the special case constructor, usually the general constructor needs to run too. Here we invoke it explicitly:

```
1 class General {
2 public:
3   General( double x,double y ) {};
4 };
5 class Special : public General {
6 public:
7   Special( double x ) : General(x,x+1) {};
8 };
```

180. Access levels

Methods and data can be

- `private`, because they are only used internally;
- `public`, because they should be usable from outside a class object, for instance in the main program;
- `protected`, because they should be usable in derived classes.

Exercise 64

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 65

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.

181. Overriding methods

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

```
1 class Base {
2 public:
3   virtual f() { ... };
4 };
5 class Deriv : public Base {
6 public:
7   virtual f() override { ... };
8 };
```

182. More

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

183. Has-a relationship

A class usually contains data members. These can be simple types or other classes. This allows you to reflect relations between things you are modeling.

```
1 class Person {
2   string name;
3   ....
4 };
5 class Course {
6 private:
7   Person the_instructor;
8   int year;
9 };`
```

This is called the has-a relation:

Course has-a *Person*

184. Literal and figurative has-a

A line segment has a starting point and an end point. *LineSegment* code design:

Store both points:

```
1 class Segment {
2 private:
3   Point p_start,p_end;
4 public:
5   Point end_point() {
6     return p_end; };
7 }
8 int main() {
9   Segment seg;
10  Point somepoint =
11    seg.end_point();
```

or store one and derive the other:

```
1 class Segment {
2 private:
3   Point starting_point;
4   float length,angle;
5 public:
6   Point end_point() {
7     /* some computation
8        from the
9        starting point */ };
10 }
```

Implementation vs API: implementation can be very different from user interface.

185. Constructors in has-a case

Class for a person:

```
class Person {
private:
  string name;
public:
  Person( string name ) {
    /* ... */
  };
};
```

Class for a course, which contains a person:

```
class Course {
private:
  Person instructor;
  int enrollment;
public:
  Course( string instr,int n )
    {
    /* ???? */
  };
};
```

Declare a `Course` variable as: `Course("Eijkhout",65);`

186. Constructors in the has-a case

Possible constructor:

```
Course( string teachername,int nstudents ) {
  instructor = Person(teachername);
  enrollment = nstudents;
};
```

Preferred:

```
Course( string teachername,int nstudents )
  : instructor(Person(teachername)),
    enrollment(nstudents) {
};
```

187. Rectangle class

Rectangle with sides parallel to the x/y axes.
Two designs possible. For the function:

```
float Rectangle::area();
```

it is most convenient to store width and height;
for inclusion testing:

```
bool Rectangle::contains(Point);
```

it would be convenient to store bottomleft/topright points.

Exercise 66

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);
   ```

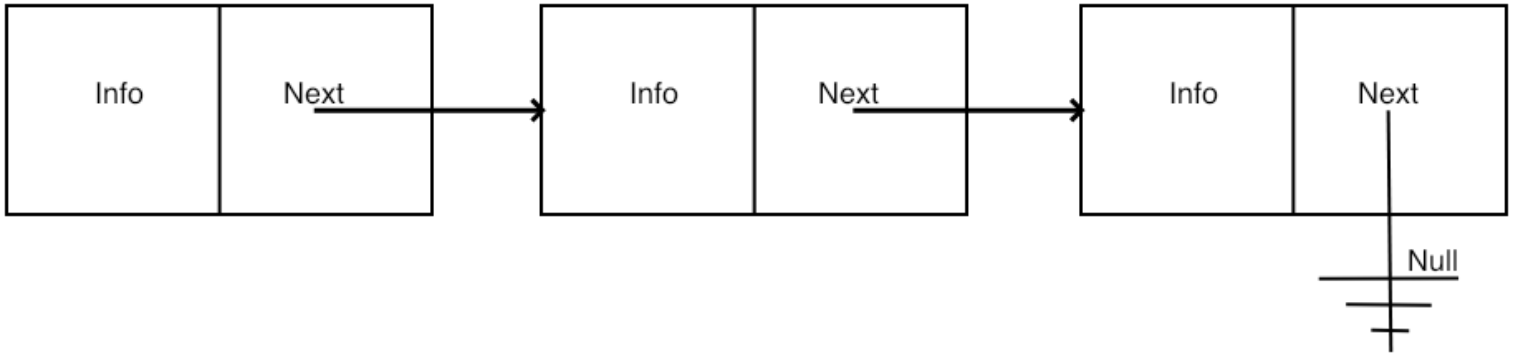
   Can you figure out how to use member initializer lists for the constructors?

Optional exercise 67

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.

Pointers

188. Motivating application: linked list



- Used inside operating systems
- Model for complicated structures: trees, DAGs.

189. Recursive data structures

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;
  /* ... */
};
```

190. Pointer types

- Smart pointers. You will see ‘shared pointers’.
- There are ‘unique pointers’. Those are tricky.
- Please don’t use old-style C pointers,
  unless you become very advanced.

191. Example: step 1, we need a class

Simple class that stores one number:

Definition:

```
// pointer/pointx.cpp
class HasX {
private:
  double x;
public:
  HasX( double x) : x(x) {};
  auto value() { return x; };
  void set(double xx) {
    x = xx; };
};
```

Example usage

```
// pointer/pointx.cpp
HasX xobj(5);
cout << xobj.value() << '\n';
xobj.set(6);
cout << xobj.value() << '\n';
```

192. Example: step 2, creating the pointer

Allocation of object and pointer to it in one:

```
auto X = make_shared<HasX>( /* args */ );

// or explicitly:

shared_ptr<HasX> X =
    make_shared<HasX>( /* constructor args */ );
```

193. Use of a shared pointer

Object vs pointed-object:

Code:

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

Output:

```
5
6
5
6
```

194. Example: step 3: headers to include

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;
```

(unique pointers will not be discussed further here)

195. Example: step 4: in use

Why do we use pointers?

Pointers make it possible for two variables to own the same object.

Code:

```
1 // pointer/pointx.cpp
2 auto xptr = make_shared<HasX>(5);
3 auto yptr = xptr;
4 cout << xptr->value() << '\n';
5 yptr->set(6);
6 cout << xptr->value() << '\n';
```

Output:

```
5
6
```

What is the difference with

```
HasX xptr(5);
HasX yptr = xptr
cout << ...stuff...
```

?

196. Pointer dereferencing

Example: function

```
float distance_to_origin( Point p );
```

How do you apply that to a `shared_ptr<Point>`?

```
shared_ptr<Point> p;
distance_to_origin( *p );
```

197. Null pointer

Initialize smart pointer to null pointer; test on null value:

```
shared_ptr<Foo> foo_ptr = nullptr;
// stuff
if (foo_ptr!=nullptr)
  foo_ptr->do_something();
```

Exercise 68

With this code given:

Code:

```
1 // pointer/dynrectangle.cpp
2 float dx( Point other ) {
3   return other.x-x; };
4   /* ... */
5   // main, with objects
6   Point
7     oneone(1,1), fivetwo(5,2);
8   float dx = oneone.dx(fivetwo);
9   /* ... */
10  // main, with pointers
11  auto
12    oneonep  = make_shared<Point>(1,1),
13    fivetwop = make_shared<Point>(5,2);
```

Output:

```
dx: 4
dx: 4
```

compute the *dx* between the *oneonep* & *fivetwop*.

You can base this off the file dynrectangle.cpp in the repository

Exercise 69

Make a *DynRectangle* class, which is constructed from two shared-pointers-to-*Point* objects:

```
// pointer/dynrectangle.cpp
auto
  origin  = make_shared<Point>(0,0),
  fivetwo = make_shared<Point>(5,2);
DynRectangle lielow( origin,fivetwo );
```

Exercise 70

Test this design: Calculate the area, scale the top-right point, and recalculate the area:

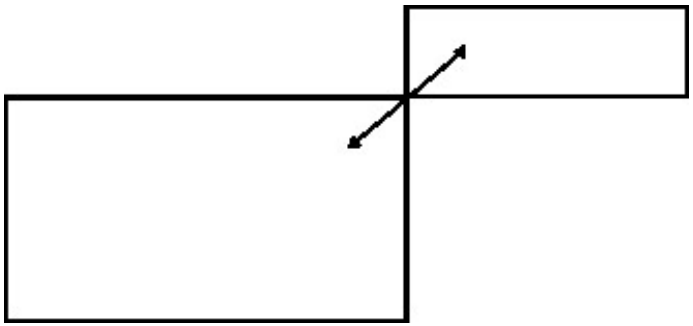
Code:

```
1 // pointer/dynrectangle.cpp
2 cout << "Area: " <<  lielow.area()
       << '\n';
3 /* ... */
4 cout << "Area: " <<  lielow.area()
       << '\n';
```

Output:

```
Area: 10
Area: 40
```

198. For the next exercise



Exercise 71

Make two *DynRectangle* objects so that the top-right corner of the first is the bottom-left corner of the other.

Now shift that point. Print out the two areas before and after to check correct behavior.

Automatic memory management

199. Memory leaks

C has a 'memory leak' problem

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

The application 'is leaking memory'.
(even worse if you do this in a loop!)
Java/Python have 'garbage collection': runtime impact
C++ has the best solution: smart pointers with reference counting.

200. Illustration

We need a class with constructor and destructor tracing:

```
// pointer/ptr1.cpp
class thing {
public:
  thing()  { cout << ".. calling constructor\n"; };
  ~thing() { cout << ".. calling destructor\n"; };
};
```

201. Show constructor / destructor in actio

Code:

```
1 // pointer/ptr0.cpp
2 cout << "Outside\n";
3 {
4   thing x;
5   cout << "create done\n";
6 }
7 cout << "back outside\n";
```

Output:

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

202. Illustration 1: pointer overwrite

Let's create a pointer and overwrite it:

Code:

```
1 // pointer/ptr1.cpp
2 cout << "set pointer1"
3      << '\n';
4 auto thing_ptr1 =
5   make_shared<thing>();
6 cout << "overwrite pointer"
7      << '\n';
8 thing_ptr1 = nullptr;
```

Output:

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

203. Illustration 2: pointer copy

Code:

```
1 // pointer/ptr2.cpp
2 cout << "set pointer2" << '\n';
3 auto thing_ptr2 =
4   make_shared<thing>();
5 cout << "set pointer3 by copy"
6      << '\n';
7 auto thing_ptr3 = thing_ptr2;
8 cout << "overwrite pointer2"
9      << '\n';
10 thing_ptr2 = nullptr;
11 cout << "overwrite pointer3"
12      << '\n';
13 thing_ptr3 = nullptr;
```

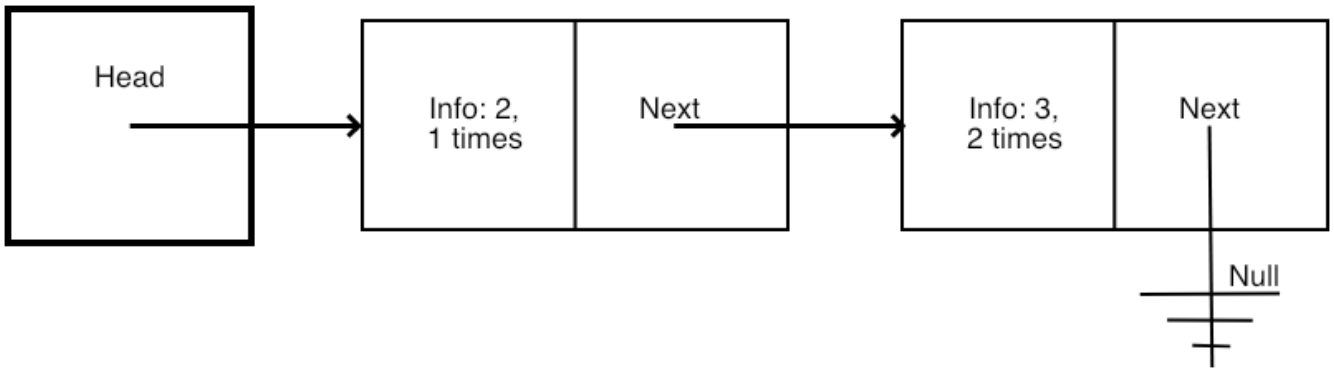
Output:

```
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.

Example: linked lists

204. Linked list



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

205. Definition of List class

A linked list has as its only member a pointer to a node:

```
// tree/linkshared.cpp
class List {
private:
  shared_ptr<Node> head{nullptr};
public:
  List() {};
```

Initially null for empty list.

206. Definition of Node class

A node has information fields, and a link to another node:

```
1  // tree/linkshared.cpp
2  class Node {
3  private:
4    int datavalue{0},datacount{0};
5    shared_ptr<Node> next{nullptr};
6  public:
7    Node() {};
8    Node(int value,shared_ptr<Node> next=nullptr)
9      : datavalue(value),datacount(1),next(next) {};
```

A Null pointer indicates the tail of the list.

207. List methods

List testing and modification.

```
List mylist;
cout << "Empty list has length: "
     << mylist.length() << '\n';

mylist.insert(3);
cout << "After one insertion the length is: "
     << mylist.length() << '\n';
if (mylist.contains_value(3))
  cout << "Indeed: contains 3" << '\n';
```

208. Recursive functions

- List structure is recursive
- Algorithms are naturally formulated recursively.

209. Recursive length computation

For the list:

```
// tree/linkshared.cpp
int List::length() {
  int count = 0;
  if (head==nullptr)
    return 0;
  else
    return head->length();
};
```

For a node:

```
// tree/linkshared.cpp
int Node::length() {
  if (!has_next())
    return 1;
  else
    return 1+next->length();
};
```

210. Iterative functions

- Recursive functions may have performance problems
- Iterative formulation possible

211. Iterative computation of the list length

Use a shared pointer to go down the list:

```
// tree/linkshared.cpp
int List::length_iterative() {
  int count = 0;
  if (head!=nullptr) {
    auto current_node = head;
    while (current_node->has_next()) {
      current_node = current_node->nextnode(); count += 1;
    }
  }
  return count;
};
```

(Fun exercise: can do an iterative de-allocate of the list?)

212. Print a list

Auxiliary function so that we can trace what we are doing.

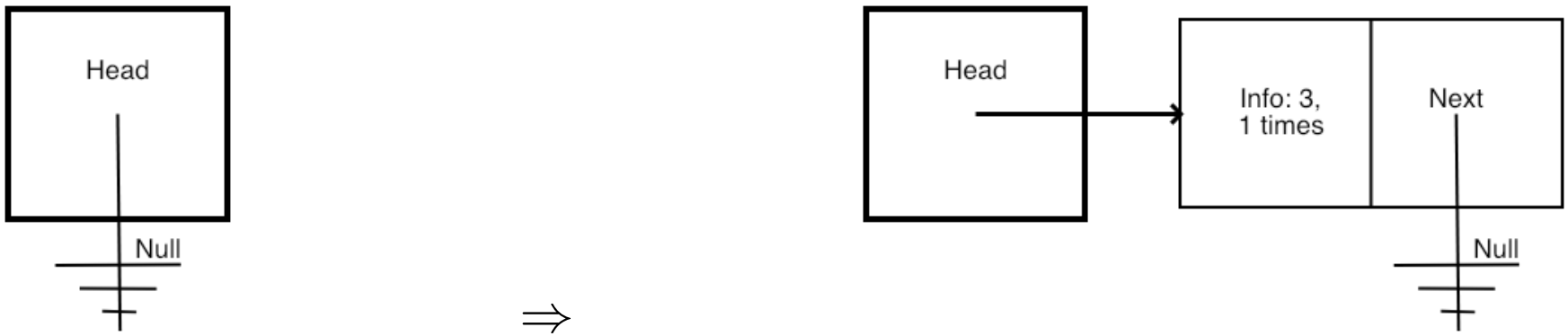
Print the list head:

```
// tree/linkshared.cpp
void List::print() {
  cout << "List:";
  if (head!=nullptr)
    cout << " => ";
    head->print();
  cout << '\n';
};
```

Print a node and its tail:

```
// tree/linkshared.cpp
void Node::print() {
  cout << datavalue << ":" <<
     datacount;
  if (has_next()) {
    cout << ", ";
    next->print();
  }
};
```

213. Creating the first list element

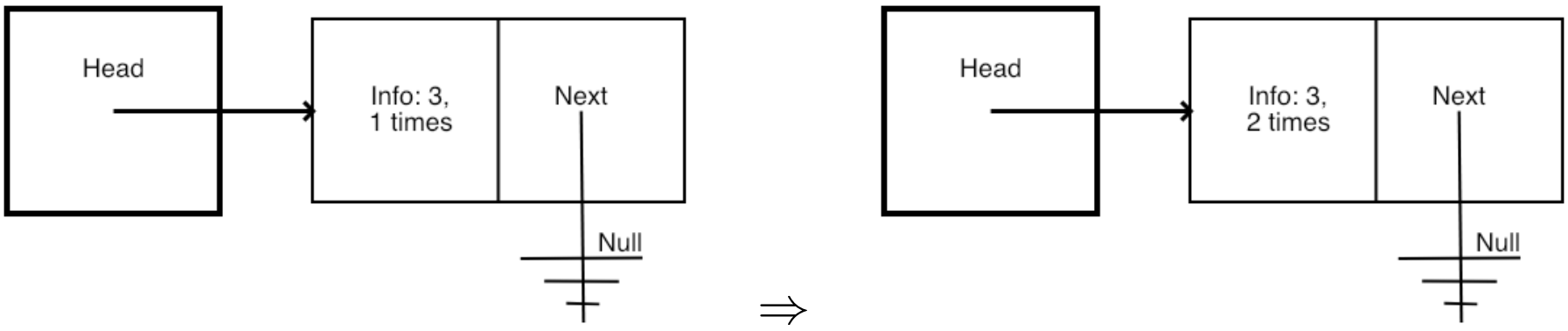


Exercise 72

Next write the case of *Node*::*insert* that handles the empty list. You also need a method *List*::*contains* that tests if an item if in the list.

```
// tree/linkshared.cpp
mylist.insert(3);
cout << "After inserting 3 the length is: "
     << mylist.length() << '\n';
if (mylist.contains_value(3))
  cout << "Indeed: contains 3" << '\n';
else
  cout << "Hm. Should contain 3" << '\n';
if (mylist.contains_value(4))
  cout << "Hm. Should not contain 4" << '\n';
else
  cout << "Indeed: does not contain 4" << '\n';
cout << '\n';
```

214. Elements that are already in the list

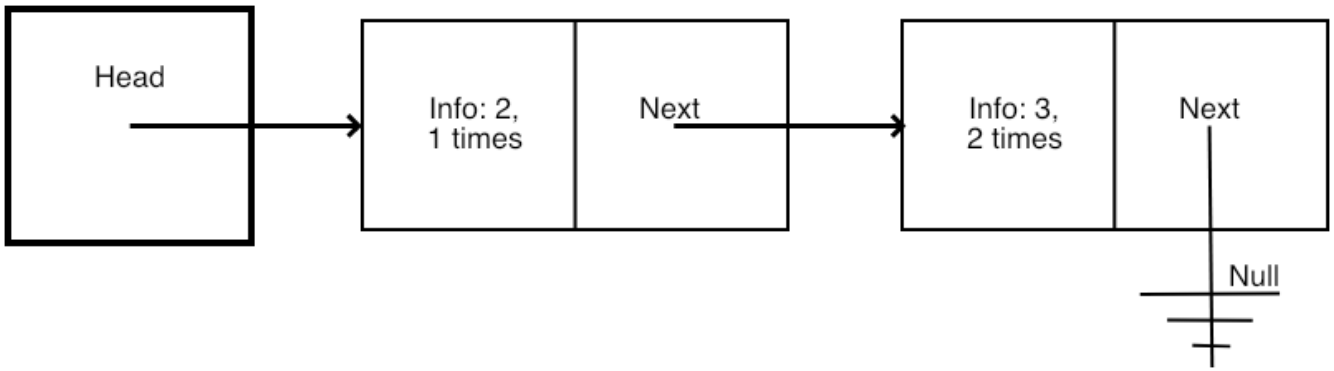


Exercise 73

Inserting a value that is already in the list means that the *count* value of a node needs to be increased. Update your *insert* method to make this code work:

```
// tree/linkshared.cpp
mylist.insert(3);
cout << "Inserting the same item gives length: "
     << mylist.length() << '\n';
if (mylist.contains_value(3)) {
  cout << "Indeed: contains 3" << '\n';
  auto headnode = mylist.headnode();
  cout << "head node has value " << headnode->value()
       << " and count " << headnode->count() << '\n';
} else
  cout << "Hm. Should contain 3" << '\n';
cout << '\n';
```

215. Element at the head

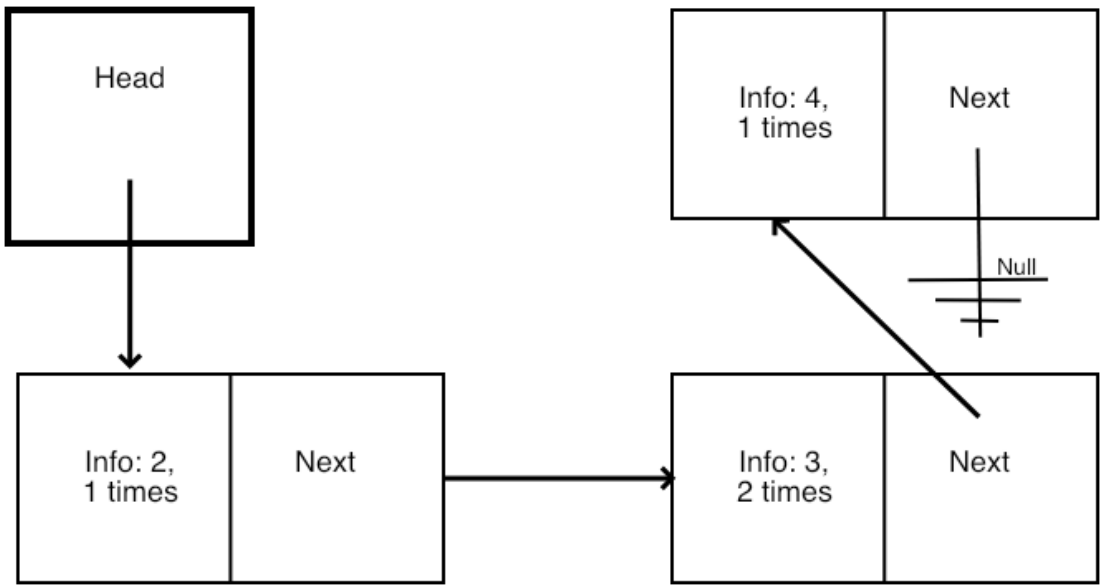


Exercise 74

One of the cases for inserting concerns an element that goes at the head. Update your *insert* method to get this to work:

```
// tree/linkshared.cpp
mylist.insert(2);
cout << "Inserting 2 goes at the head;\nnow the length is: "
     << mylist.length() << '\n';
if (mylist.contains_value(2))
  cout << "Indeed: contains 2" << '\n';
else
  cout << "Hm. Should contain 2" << '\n';
if (mylist.contains_value(3))
  cout << "Indeed: contains 3" << '\n';
else
  cout << "Hm. Should contain 3" << '\n';
cout << '\n';
```

216. Element at the tail

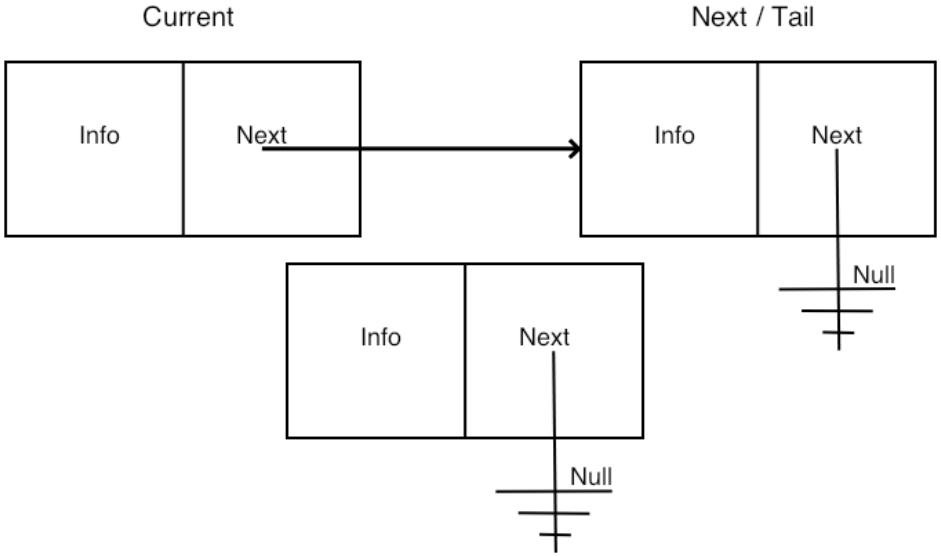


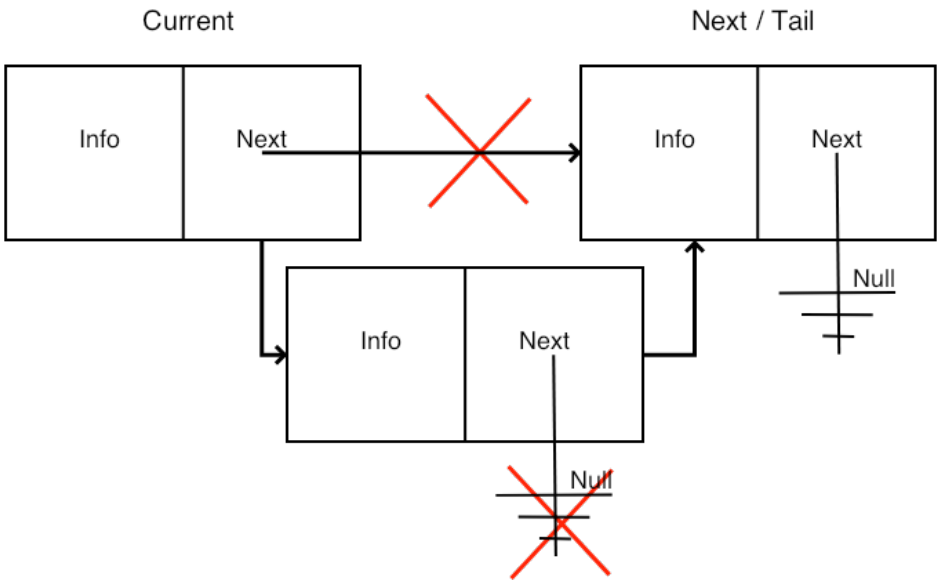
Exercise 75

If an item goes at the end of the list:

```
// tree/linkshared.cpp
mylist.insert(6);
cout << "Inserting 6 goes at the tail;\nnow the length is: "
     << mylist.length()
     << '\n';
if (mylist.contains_value(6))
  cout << "Indeed: contains 6" << '\n';
else
  cout << "Hm. Should contain 6" << '\n';
if (mylist.contains_value(3))
  cout << "Indeed: contains 3" << '\n';
else
  cout << "Hm. Should contain 3" << '\n';
cout << '\n';
```

217. Insertion





Exercise 76

Update your insert routine to deal with elements that need to go somewhere in the middle.

```
// tree/linkshared.cpp
mylist.insert(4);
cout << "Inserting 4 goes in the middle;\nnow the length is: "
     << mylist.length()
     << '\n';
if (mylist.contains_value(4))
  cout << "Indeed: contains 4" << '\n';
else
  cout << "Hm. Should contain 4" << '\n';
if (mylist.contains_value(3))
  cout << "Indeed: contains 3" << '\n';
else
  cout << "Hm. Should contain 3" << '\n';
cout << '\n';
```

218. Linked list exercise

Write a program that constructs a linked list where the elements are sorted in increasing numerical order.

Your program should accept a sequence of numbers from interactive input, and after each number print the list for as far as it has been constructed. Print the list on a single line, with elements separated by commas.

An input value of zero signals the end of input; this number is not added to the list.

Pointers and addresses

C and F pointers

C++ and Fortran have a clean reference/pointer concept: a reference or pointer is an 'alias' of the original object

C/C++ also has a very basic pointer concept:
a pointer is the address of some object
(including pointers)

If you're writing C++ you should not use it.
(until you get pretty advanced)
if you write C, you'd better understand it.

219. Memory addresses

If you have an `int i`, then `&i` is the address of `i`.

An address is a (long) integer, denoting a memory address. Usually it is rendered in hexadecimal notation.

Code:

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

Output:

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

220. Same in C

Using purely C:

Code:

```
1 // pointer/printfpoint.cpp
2 int i;
3 printf("address of i: %ld\n",
4         (long)(&i));
5 printf(" same in hex: %lx\n",
6         (long)(&i));
```

Output:

```
address of i:
     140732690693076
 same in hex:
     7ffee2097bd4
```

221. Address types

The type of '`&i`' is `int*`, pronounced 'int-star', or more formally: 'pointer-to-int'.

You can create variables of this type:

```
int i;
int* addr = &i;
// exactly the same:
int *addr = &i;
```

Now `addr` contains the memory address of `i`.

222. Dereferencing

Using **addr* 'dereferences' the pointer: gives the thing it points to; the value of what is in the memory location.

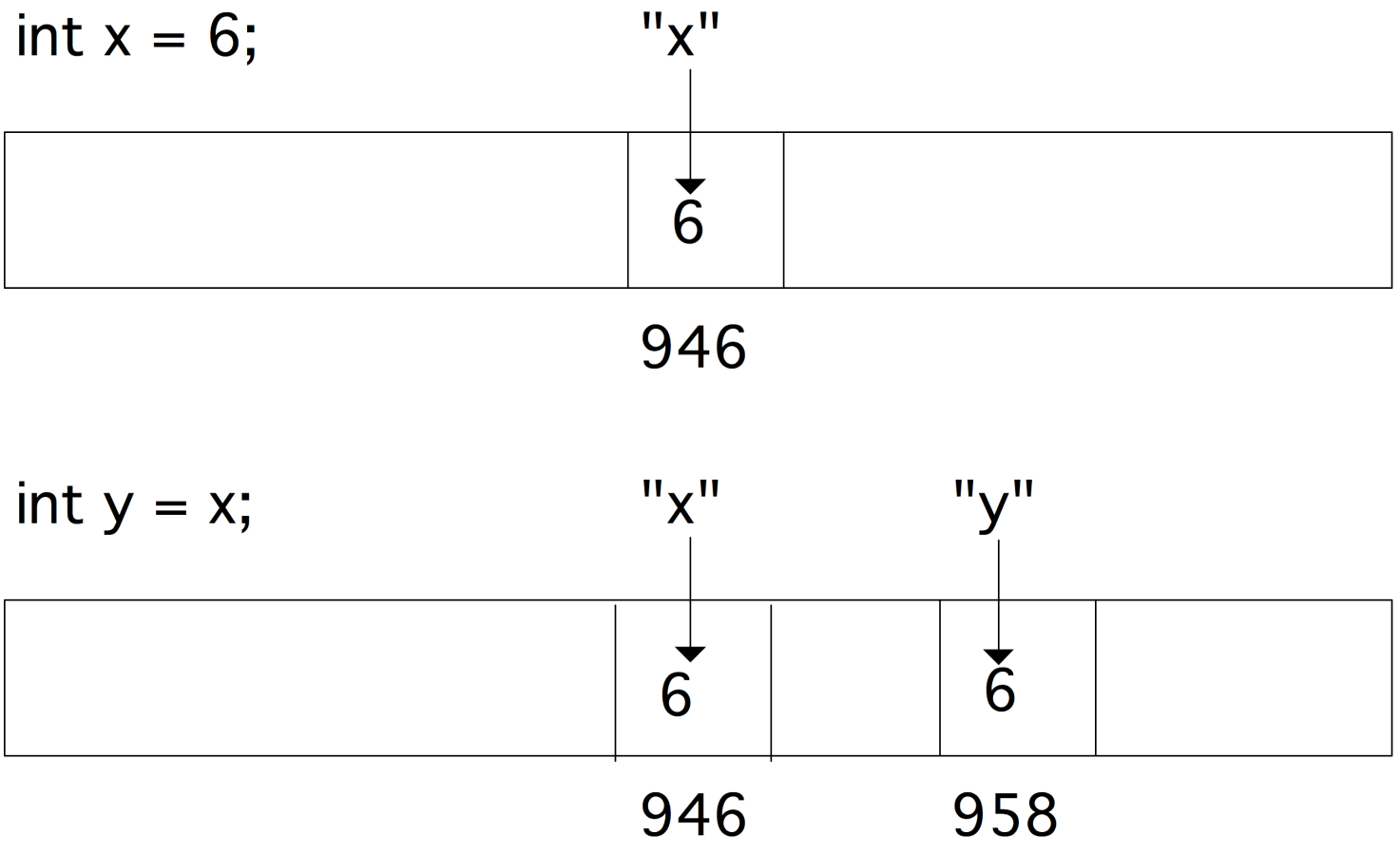
Code:

```
1 // pointer/cintpointer.cpp
2 int i;
3 int* addr = &i;
4 i = 5;
5 cout << *addr << '\n';
6 i = 6;
7 cout << *addr << '\n';
```

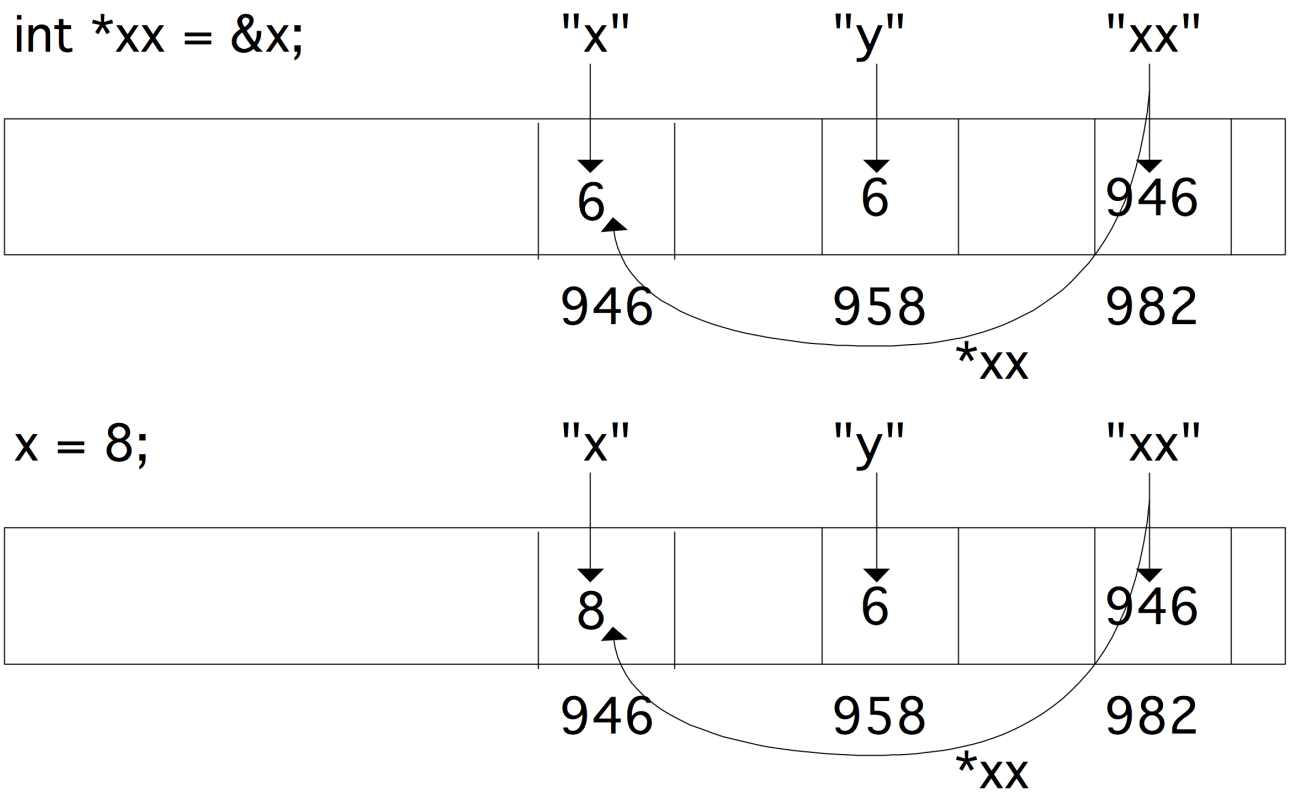
Output:

```
5
6
```

223. illustration



224. illustration



225. Star stuff

Equivalent:

- `int* addr`: `addr` is an int-star, or
- `int *addr`: `*addr` is an int.

Addresses and parameter passing

226. C++ pass by reference

C++ style functions that alter their arguments:

```
void inc(int &i) {
  i += 1;
}
int main() {
  int i=1;
  inc(i);
  cout << i << endl;
  return 0;
}
```

227. C-style pass by reference

In C you can not pass-by-reference like this. Instead, you pass the address of the variable `i` by value:

```
void inc(int *i) {
  *i += 1;
}
int main() {
  int i=1;
  inc(&i);
  cout << i << endl;
  return 0;
}
```

Now the function gets an argument that is a memory address: `i` is an int-star. It then increases `*i`, which is an int variable, by one.

Exercise 77

Write another version of the *swap* function:

```
void swap( /* something with i and j */ {
  /* your code */
}
int main() {
  int i=1,j=2;
  swap( /* something with i and j */ );
  cout << "check that i is 2: " << i << endl;
  cout << "check that j is 1: " << i << endl;
  return 0;
}
```

Hint: write C++ code, then insert stars where needed.

Arrays and pointers

228. Array and pointer equivalence

Array and memory locations are largely the same:

Code:

```
1 // pointer/arrayaddr.cpp
2 double array[5] = {11,22,33,44,55};
3 double *addr_of_second =
        &(array[1]);
4 cout << *addr_of_second << '\n';
5 array[1] = 7.77;
6 cout << *addr_of_second << '\n';
```

Output:

```
22
7.77
```

229. Array passing to function

When an array is passed to a function, it behaves as an address:

Code:

```
1 // pointer/arraypass.cpp
2 void set_array( double *x,int size)
      {
3   for (int i=0; i<size; ++i)
4     x[i] = 1.41;
5 };
6     /* ... */
7   double array[5] =
      {11,22,33,44,55};
8   set_array(array,5);
9   cout << array[0] << "...." <<
      array[4] << '\n';
```

Output:

```
1.41....1.41
```

Note that these arrays don't know their size, so you need to pass it.

230. Size of an array

There is a `sizeof` function but beware:

Code:

```
1 // c/carray.c
2 void stat_f( int stat[] ) {
3   printf(".. in function:
       %lu\n",sizeof(stat));
4 }
5     /* ... */
6   int stat[23];
7   printf("Size of stat[23]:
       %lu\n",sizeof(stat));
8   stat_f( stat );
```

Output:

```
Size of stat[23]: 92
.. in function: 8
```

(This is an example of pointer decay)

Multi-dimensional arrays

231. Multi-dimensional arrays

After

```
double x[10][20];
```

a row `x[3]` is a `double*`, so is `x` a `double**`?

Was it created as:

```
double **x = new double*[10];
for (int i=0; i<10; i++)
  x[i] = new double[20];
```

No: multi-d arrays are contiguous.

Dynamic allocation

232. Problem with static arrays

Create an array with size depending on something:

```
if ( something ) {
  double ar[25];
} else {
  double ar[26];
}
ar[0] = // there is no array!
```

This Does Not Work

233. Declaration and allocation

Now dynamic allocation:

```
double *array;
if (something) {
  array = new double[25];
} else {
  array = new double[26];
}
```

Don't forget:

```
delete array;
```

234. Allocation, C vs C++

C allocates in bytes:

```
double *array;
array  = (double*) malloc( 25*sizeof(double) );
```

C++ allocates an array:

```
double *array;
array  = new double[25];
```

Don't forget:

```
free(array); // C
delete array; // C++
```

235. De-allocation

Memory allocated with *malloc* / new does not disappear when you leave a scope. Therefore you have to delete the memory explicitly:

```
free(array);
delete(array);
```

The C++ *vector* does not have this problem, because it obeys scope rules.

236. Memory leak1

```
void func() {
  double *array = new double[large_number];
  // code that uses array
}
int main() {
  func();
};
```

- The function allocates memory
- After the function ends, there is no way to get at that memory
- $\Rightarrow$ memory leak.

237. Memory leaks

```
for (int i=0; i<large_num; i++) {
  double *array = new double[1000];
  // code that uses array
}
```

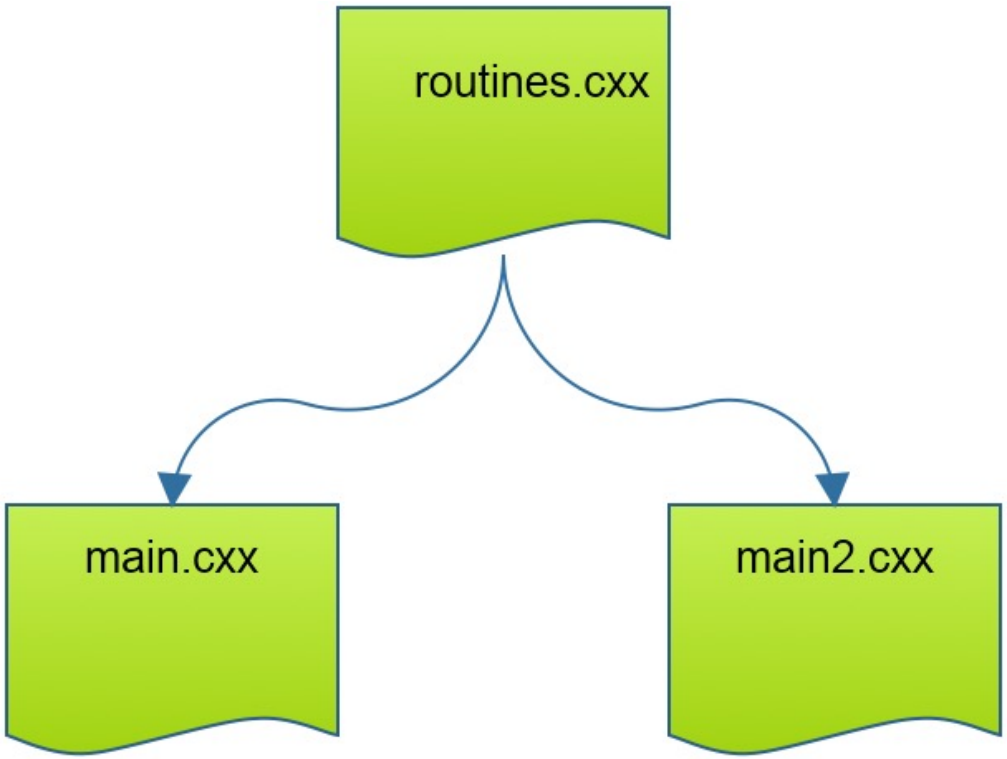
Every iteration reserves memory, which is never released: another memory leak.

Your code will run out of memory!

Prototypes, separate compilation

238. Include files

- Code reuse is good.
- How would you use functions/classes in more than one main?



We will discuss systematic solutions.

239. Reminder: definition vs declaration

Definition:

```
bool iseven( int n ) { return n%2==0 }
```

Declaration:

```
bool iseven( int n );
// or even:
bool iseven( int );
```

240. Declarations, case 1

Some people like defining functions after the main.
Problem: the main needs to know about them.

'forward declaration'

```
int f(int);
int main() {
  f(5);
};
int f(int i) {
  return i;
}
```

versus:

```
int f(int i) {
  return i;
}
int main() {
  f(5);
};
```

This is a stylistic choice.

241. Declarations, case 2

You also need forward declaration for mutually recursive functions:

```
int f(int);
int g(int i) { return f(i); }
int f(int i) { return g(i); }
```

242. Separate compilation

Split your program in multiple files.

- Easier to edit
- Less chance of `git` conflicts
- Only recompile the file you edit
  $\Rightarrow$ reduction of compile/build time.

243. Declarations for separate compilation

Define a function in one file;
an other file uses it, so needs the declaration:

```
// file: def.cpp
int tester(float x) {
  .....
}
```

```
// file : main.cpp
int tester(float);

int main() {
  int t = tester(...);
  return 0;
}
```

This Is Not A Good Design!

244. Declarations and header files

Using a header file with function declarations.

Header file contains only declaration:

```
// file: def.h
int tester(float);
```

The header file gets included both in the definitions file and the main program:

```
// file: def.cpp
#include "def.h"
int tester(float x) {
  .....
}
```

```
// file : main.cpp
#include "def.h"

int main() {
  int t = tester(...);
  return 0;
}
```

What happens if you leave out the `#include "def.h"` in both cases?

245. Class declarations

Header file:

```
// proto/functheader.hpp
class something {
private:
  int i;
public:
  double dosomething( int i, char c );
};
```

Implementation file: **missing snippet classheaderimpl**

246. File naming convention

- Source files: `.cpp` `.cxx`
  I use `.cpp` for no real reason
- Header files: `.h` `.hpp` `.hxx`
  I use `.hpp` by analogy with `.cpp`
  `.h` reminds me too much of C.

247. Compiling and linking

Your regular compile line

```
icpc -o yourprogram yourfile.cc
```

actually does two things: compilation, and linking. You can do those separately:

1. First you compile

   ```
   icpc -c yourfile.cc
   ```

   which gives you a file `yourfile.o`, a so-called object file; and
2. Then you use the compiler as linker to give you the executable file:

   ```
   icpc -o yourprogram yourfile.o
   ```

248. Dealing with multiple files

Compile each file separately, then link:

```
icpc -c mainfile.cc
icpc -c functionfile.cc
icpc -o yourprogram mainfile.o functionfile.o
```

249. Header file with include guard

Header file tests if it has already been included:

```
// this is foo.h
#ifndef FOO_H
#define FOO_H

// the things that you want to include

#endif
```

This prevents double or recursive inclusion.

250. Make

Good idea to learn the `Make` utility for project management.

(Also `Cmake`.)

251. Skeleton example

Directory `skeletons/funct_skeleton` contains

```
funct.cpp functheader.hpp functmain.cpp
```

CMake setup:

```
add_executable(
    funct functmain.cpp funct.cpp functheader.hpp )
```

252. CMake compilation

Do `cmake` and then `make`:

```
[ 33%] Building CXX object CMakeFiles/funct.dir/functmain.cpp.o
[ 66%] Building CXX object CMakeFiles/funct.dir/funct.cpp.o
[100%] Linking CXX executable funct
[100%] Built target funct
```

253. Justification for separate compilation

- Edit only `funct.cpp`;
- Do *not* `cmake`;
- do `make`

```
( cd build && make )
Consolidate compiler generated dependencies of target funct
[ 33%] Building CXX object CMakeFiles/funct.dir/funct.cpp.o
[ 66%] Linking CXX executable funct
[100%] Built target funct
```

Only that file got recompiled.

Advanced topics

254. Why lambda expressions?

Lambda expressions (sometimes incorrectly called 'closures') are 'anonymous functions'. Why are they needed?

- Small functions may be needed; defining them is tedious, would be nice to just write the function recipe in-place.
- C++ can not define a function dynamically, depending on context.
  Example:
  1. we read `float c`
  2. now we want function `float f(float)` that multiplies by `c`:

```
float c; cin >> c;
float mult( float x ) { // DOES NOT WORK
   // multiply x by c
};
```

255. Introducing: lambda expressions

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 // lambda/lambdaex.cpp
2 [] (float x,float y) -> float {
3   return x+y; } ( 1.5, 2.3 )
```

Output:

```
3.8
```

256. Lambda syntax

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

- The square brackets are how you recognize a lambda; we will get to the 'capture' later. For now it will often be empty.
- Inputs: like function parameters
- Result type specification `-> outtype`: can be omitted if compiler can deduce it;
- Definition: function body.

257. Assign lambda expression to variable

Code:

```
1 // lambda/lambdaex.cpp
2 auto summing =
3   [] (float x,float y) -> float {
4   return x+y; };
5 cout << summing ( 1.5, 2.3 ) << '\n';
6 cout << summing ( 3.7, 5.2 ) << '\n';
```

Output:

```
3.8
8.9
```

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

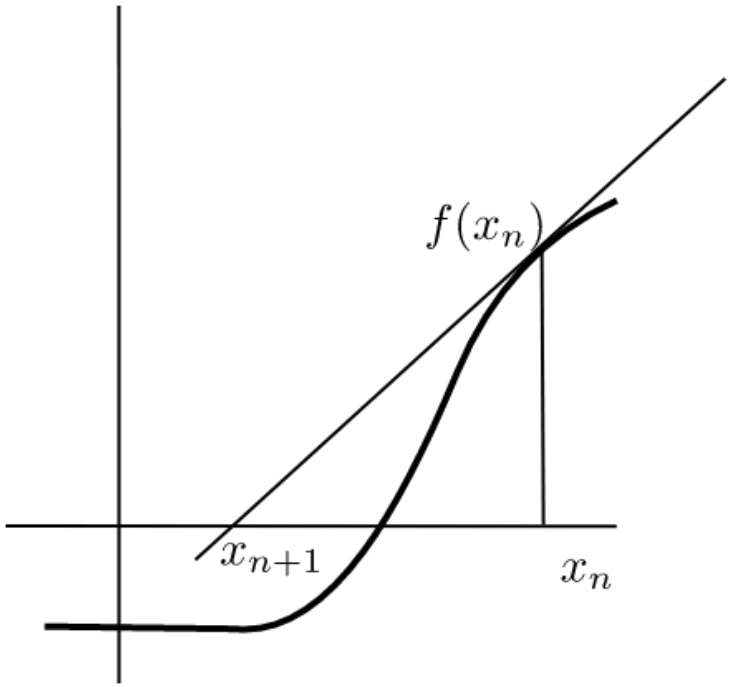
Return type could have been omitted:

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

Example of lambda usage: Newton's method

258. Newton's method

$$x_{n+1} = x_n - f(x_n)/f'(x_n)$$



259. Newton for root finding

With

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

zero finding is equivalent to

$$f(x) = 0 \quad \text{for } x = \sqrt{2}$$

so we can compute a square root if we have a zero-finding function.

Newton's method for this f:

$$x_{n+1} = x_n - f(x_n)/f'(x_n) = x_n - \frac{(x_n^2 - 2)}{2x_n} = x_n/2 + 2/x_n$$

Square root computation only takes division!

Exercise 78

Rewrite your code to use lambda functions for `f` and `fprime`.

If you use variables for the lambda expressions, put them in the main program.

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

260. Function pointers

You can pass a function to another function.
In C syntax:

```
1  void f(int i) { /* something with i */ };
2  void apply_to_5( (void)(*f)(int) ) {
3      f(5);
4  }
5  int main() {
6    apply_to_5(f);
7  }
```

(You don't have to understand this syntax. The point is that you can pass a function as argument.)

261. Lambdas as parameter: the problem

Lambdas have a type that is dynamically generated, so you can not write a function that takes a lambda as argument, because you can't write the type.

```
1  void apply_to_5( /* what? */ f ) {
2      f(5);
3  }
4  int main() {
5      apply_to_5
6        ( [] (double x) { cout << x; } );
7  }
```

262. Lambdas as parameter: the solution

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

With this, you can declare function parameters by their signature (that is, types of parameters and output):

Code:

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

Output:

```
Int: 5
```

263. Lambdas expressions for Newton

We are going to write a Newton function which takes two parameters: an objective function, and its derivative; it has a `double` as result.

```
// newton/newton-lambda.cpp
double newton_root
    ( function< double(double) > f,
      function< double(double) > fprime ) {
```

This states that `f,fprime` are in the class of `double(double)` functions: `double` parameter in, `double` result out.

Exercise 79

Rewrite the Newton exercise by implementing a *newton_root* function:

```
double root = newton_root( f,fprime );
```

Call the function

1. first with the lambda variables you already created;
2. then directly with the lambda expressions as arguments, that is, without assigning them to variables.

Captures

264. Capture variable

Increment function:

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

Code:

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

Output:

```
6
13
26
```

265. Capture more than one variable

Example: multiply by a fraction.

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

Exercise 80

- 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?)

266. Capture value is copied

Illustrating that the capture variable is copied once and for all:

Code:

```
1 // lambda/lambdacapture.cpp
2 int inc;
3 cin >> inc;
4 auto increment =
5   [inc] ( int input ) -> int {
6     return input+inc;
7   };
8 cout << "increment by: " << inc << '\n';
9 cout << "1 -> "
10      << increment(1) << '\n';
11 inc = 2*inc;
12 cout << "1 -> "
13      << increment(1) << '\n';
```

Output:

```
increment by: 2
1 -> 3
1 -> 3
```

Exercise 81

Extend the newton exercise to compute roots in a loop:

```
// newton/newton-lambda.cpp
  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.

267. Derivative by finite difference

You can approximate the derivative of a function f as

$$f'(x) = \big(f(x+h) - f(x)\big)/h$$

where h is small.
This is called a 'finite difference' approximation.

Exercise 82

Write a version of the root finding function that only takes the objective function:

```
double newton_root( function< double(double)> f )
```

and approximates the derivative by a finite difference. 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.
This is polymorphism: you now have two definition for the same function. They differ in the number of arguments.

268. Turn it in!

Write a program that

1. reads an integer from the commandline
2. prints a line:
   `The root of this number is 1.4142`
   which contains the word `root` and the value of the square root of the input in default output format.

Your program should

- have a subroutine `newton_root` as described above.
- (8/10 credit): call it with two lambda expressions: one for the function and one for the derivative, *or*
- (10/10 credit) call it with a single lambda expression for the function and approximate the derivative as described above.

269. Lambda in object

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

```
// lambda/lambdafun.cpp
#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;
  };
};
```

270. Illustration

The above code in use:

Code:

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

Output:

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

Advanced topics

271. Capture by value

Normal capture is by value:

Code:

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

Output:

```
6
13
26
```

272. Capture by reference

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 **??**.)

273. Lambdas vs function pointers

Lambda expression with empty capture are compatible with C-style function pointers:

Code:

```
1 // lambda/lambdacptr.cpp
2 int cfun_add1( int i ) {
3   return i+1; };
4 int apply_to_5( int(*f)(int) ) {
5   return f(5); };
6 //codesnippet  end
7     /* ... */
8   auto lambda_add1 =
9     [] (int i) { return i+1; };
10  cout << "C ptr: "
11       << apply_to_5(&cfun_add1)
12       << '\n';
13  cout << "Lambda: "
14       << apply_to_5(lambda_add1)
15       << '\n';
```

Output:

```
missing snippet
../../code/func/lambdacptr.runout
```

274. Use in algorithms

```
for_each( myarray, [] (int i) { cout << i; } );

transform( myarray, [] (int i) { return i+1; } );
```

See later.

Namespaces

275. Namespaces in action

How do you indicate that something comes from a namespace?

Option: explicitly indicated.

```
#include <vector>
int main() {
  std::vector<stuff> foo;
}
```

Import the whole namespace:

```
#include <vector>
using namespace std;
int main() {
  vector<stuff> foo;
}
```

Good compromise:

```
#include <vector>
using std::vector;
int main() {
  vector<stuff> foo;
}
```

276. Why not 'using namespace std'?

Illustrating the dangers of `using namespace std`:

This compiles, but should not:

```
// func/swapname.cpp
#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;
}
```

(Why?)

This gives an error:

```
// func/swapusing.cpp
#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;
}
```

277. Defining a namespace

Introduce new namespace:

```
namespace geometry {
  // definitions
  class vector {
  };
|
```

278. Namespace usage

Double-colon notation for namespace and type:

```
geometry::vector myobject();
```

or

```
using geometry::vector;
vector myobject();
```

or even

```
using namespace geometry;
vector myobject();
```

Exceptions

279. Throw an integer

Throw an integer error code:

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

280. Catching an exception

Catch an integer:

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

281. Exception classes

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

throw( MyError(27,"oops");

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

You can use exception inheritance!

282. Multiple catches

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

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

283. Catch any exception

Catch exceptions without specifying the type:

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

284. More about exceptions

- Functions can define what exceptions they throw:

```
void func() throw( MyError, std::string );
void funk() throw();
```

- Predefined exceptions: `bad_alloc`, `bad_exception`, etc.
- An exception handler can throw an exception; to rethrow the same exception use '`throw;`' without arguments.
- Exceptions delete all stack data, but not `new` data. Also, destructors are called; section **??**.
- 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`.

285. Destructors and exceptions

The destructor is called when you throw an exception:

Code:

```
1  // object/exceptdestruct.cpp
2  class SomeObject {
3  public:
4    SomeObject() {
5      cout << "calling the
        constructor"
6           << '\n'; };
7    ~SomeObject() {
8      cout <<  "calling the
        destructor"
9           << '\n'; };
10 };
11     /* ... */
12   try {
13     SomeObject obj;
14     cout << "Inside the nested
        scope" << '\n';
15     throw(1);
16   } catch (...) {
17     cout << "Exception caught" <<
        '\n';
```

Output:

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

Auto

286. Type deduction

In:

```
std::vector< std::shared_ptr< myclass >>*
  myvar = new std::vector< std::shared_ptr< myclass >>
                ( 20, new myclass(1.3) );
```

the compiler can figure it out:

```
auto myvar =
  new std::vector< std::shared_ptr< myclass >>
            ( 20, new myclass(1.3) );
auto result = someobject.somemethod();
```

287. Type deduction in functions

Return type of functions can be deduced in C++17:

```
// auto/autofun.cpp
auto equal(int i,int j) {
  return i==j;
};
```

288. Auto and references, 1

Demostrating that `auto` discards references from the rhs:

Code:

```
1 // auto/plainget.cpp
2 A my_a(5.7);
3 // reminder: float& A::access()
4 auto get_data = my_a.access();
5 get_data += 1;
6 my_a.print();
```

Output:

```
data: 5.7
```

289. Auto and references, 2

Combine `auto` and references:

Code:

```
1 // auto/refget.cpp
2 A my_a(5.7);
3 auto &get_data = my_a.access();
4 get_data += 1;
5 my_a.print();
```

Output:

```
data: 6.7
```

290. Auto and references, 3

For good measure:

```
1 // auto/constrefget.cpp
2 A my_a(5.7);
3 const auto &get_data = my_a.access();
4 get_data += 1; // WRONG does not compile
5 my_a.print();
```

291. Ranges vs iterators

Equivalence of range and iterator code:

The range code

```
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!

Random

292. Random floats

Random numbers from the unit interval:

```
// rand/xrand.cpp
// seed the generator
std::random_device r;
// set the default random number generator
std::default_random_engine generator{r()};
// distribution: real between 0 and 1
std::uniform_real_distribution<float> distribution(0.,1.);

for ( int i=0; i<5; i++)
  cout << "random: "
       << distribution(generator)
       << '\n';
```

293. Dice throw

```
// set the default generator
std::default_random_engine generator;

// distribution: ints 1..6
std::uniform_int_distribution<int> distribution(1,6);

// apply distribution to generator:
int dice_roll = distribution(generator);
  // generates number in the range 1..6
```

294. Poisson distribution

Poisson distributed integers:
chance of k occurrences, if m is the average number
(or $1/m$ the probability)

```
std::default_random_engine generator;
float mean = 3.5;
std::poisson_distribution<int> distribution(mean);
int number = distribution(generator);
```

295. Global engine

Good approach: random generator static in the function.

Code:

```
1 // rand/static.cpp
2 int realrandom_int(int max) {
3   static
       std::default_random_engine
       static_engine;
4   std::uniform_int_distribution<>
       ints(1,max);
5   return ints(static_engine);
6 };
```

Output:

```
Three ints: 15, 98, 70.
```

A single instance is ever created.

Other stuff

296. Static variables

Static variable exists once per class, not per object:

```
class Thing {
private:
  static inline int n_things=0; // global count
  int mynumber; // who am I?
```

increase in constructor:

```
Thing::Thing() {
  mynumber = n_things++; };
```

Libraries

Software development

Intro to testing

297. Dijkstra quote

> *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)*

Still ...

298. Types of testing

- *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.

299. Unit testing

- Every part of a program should be testable
- $\Rightarrow$ good idea to have a function for each bit of functionality
- Positive tests: show that code works when it should
- Negative tests: show that the code fails when it should

300. Unit testing

- Every part of a program should be testable
- Do not write the tests after the program:
  write tests while you develop the program.
- Test-driven development:
  1. design functionality
  2. write test
  3. write code that makes the test work

301. Principles of TDD

Develop code and tests hand-in-hand:

- 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.

302. Unit testing frameworks

Testing is important, so there is much software to assist you.

Popular choice with C++ programmers: Catch2
`https://github.com/catchorg`

Intro to Catch2

303. Toy example

Function and tester:

```
// catch/require.cpp
#define CATCH_CONFIG_MAIN
#include "catch2/catch_all.hpp"

int five() { return 5; }

TEST_CASE( "needs to be 5" ) {
    REQUIRE( five()==5 );
}
```

The define line supplies a main:
you don't have to write one.

304. Tests that fail

```
// catch/require.cpp
float fiveish() { return 5.00001; }
TEST_CASE( "not six" ) {
  // this will fail
  REQUIRE( fivish()==5 );
  // this will succeed
  REQUIRE( fivish()==Catch::Approx(5) );
}
```

Exercise 83

Write a function *is_prime*, and write a test case for it. This should have both cases that succeed and that fail.

305. Boolean tests

Test a boolean expression:

```
REQUIRE( some_test(some_input) );
REQUIRE( not some_test(other_input) );
```

306. Output for failing tests

Run the tester:

Code:

```
1 // catch/false.cpp
2 #define CATCH_CONFIG_MAIN
3 #include "catch2/catch_all.hpp"
4
5 int five() { return 6; }
6
7 TEST_CASE( "needs to be 5" ) {
8     REQUIRE( five()==5 );
9 }
```

Output:

```
Randomness seeded to:
     235485692

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
false is a Catch2
     v3.1.1 host
     application.
Run with -? for options

-------------------------------------------
needs to be 5
-------------------------------------------
false.cpp:21
...........................................

false.cpp:22: FAILED:
  REQUIRE( five()==5 )
with expansion:
  6 == 5

===========================================
```

307. Diagnostic information for failing tests

INFO: print out information at a failing test

```
TEST_CASE( "test that f always returns positive" ) {
  for (int n=0; n<1000; n++)
    INFO( "iteration: " << n );
    REQUIRE( f(n)>0 );
}
```

308. Exceptions

Exceptions are a mechanism for reporting an error:

```
double SquareRoot( double x ) {
  if (x<0) throw(1);
  return std::sqrt(x);
};
```

More about exceptions later;
for now: Catch2 can deal with them

309. Test for exceptions

Suppose a function $g(n)$ satisfies:

- it succeeds for input $n > 0$
- it 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) );
}
```

310. Slightly realistic example

We want a function that

- computes a square root for $x \geq 0$
- throws an exception for $x < 0$;

```
// catch/sqrt.cpp
double root(double x) {
  if (x<0) throw(1);
  return std::sqrt(x);
};

TEST_CASE( "test sqrt function" ) {
  double x=3.1415, y;
  REQUIRE_NOTHROW( y=root(x) );
  REQUIRE( y*y==Catch::Approx(x) );
  REQUIRE_THROWS( y=root( -3.14 ) );
}
```

What happens if you require:

```
  REQUIRE( y*y==x );
```

311. Tests with code in common

Use *SECTION* if tests have intro/outtro in common:

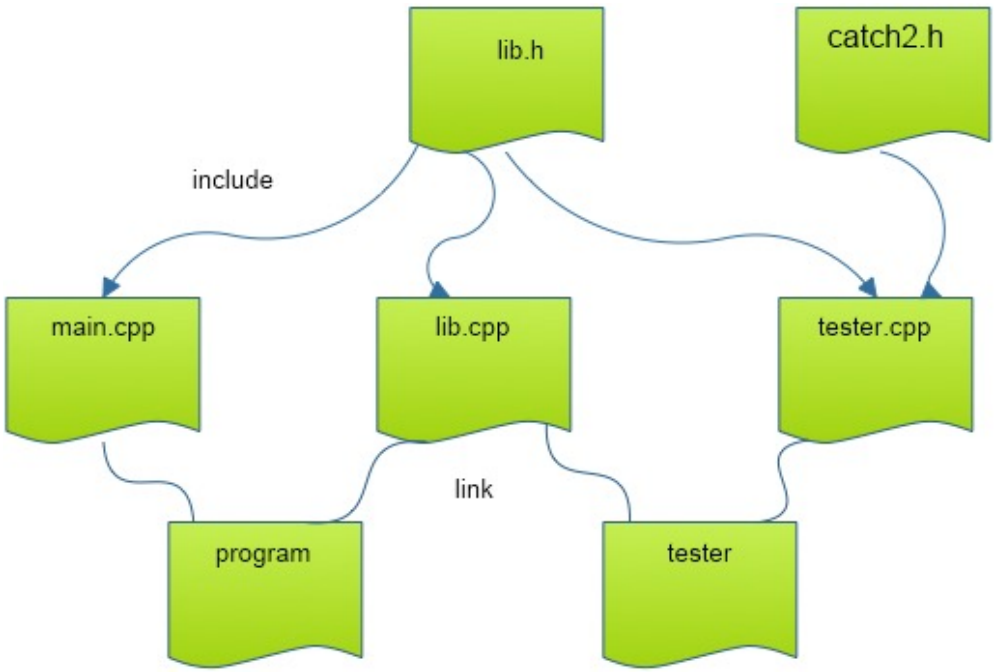
```
TEST_CASE( "commonalities" ) {
  // common setup:
  double x,y,z;
  REQUIRE_NOTHROW( y = f(x) );
  // two independent tests:
  SECTION( "g function" ) {
    REQUIRE_NOTHROW( z = g(y) );
  }
  SECTION( "h function" ) {
    REQUIRE_NOTHROW( z = h(y) );
  }
  // common followup
  REQUIRE( z>x );
}
```

(sometimes called setup/teardown)

Catch2 file structure

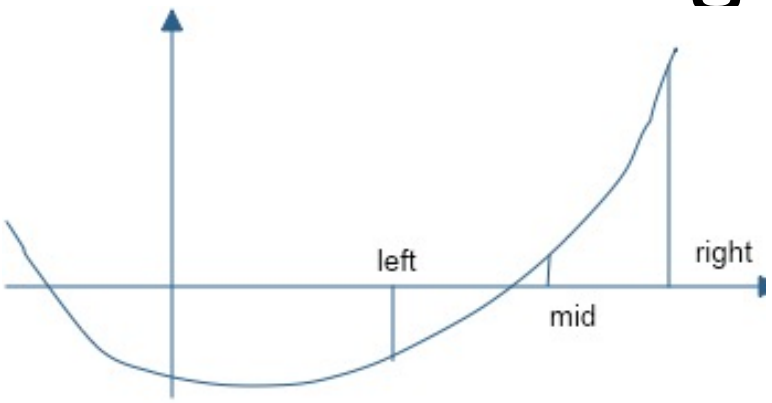
312. Realistic setup

- All program functionality in a 'library':
  split between header and implementation
- Main program can be short
- Tester file with only tests.
- (Tester also needs the `catch2` stuff included)



TDD example: Bisection

313. Root finding by bisection



- Start with bounds where the function has opposite signs.

$$x_- < x_+, \qquad f(x_-) \cdot f(x_+) < 0,$$

- Find the mid point;
- Adjust either left or right bound.

314. Coefficient handling

$$f(x) = c_0 x^d + c_1 x^{d-1} \cdots + c_{d-1} x^1 + c_d$$

We implement this by constructing a *polynomial* object from coefficients in a *vector*<double>:

```
// bisect/zeroclasslib.hpp
class polynomial {
private:
  std::vector<double> coefficients;
public:
  polynomial( std::vector<double> c );
```

Exercise 84: Test for proper coefficients

For polynomial coefficients to give a well-defined polynomial, the zero-th coefficient needs to be non-zero:

```
// bisect/zeroclasstest.cpp
TEST_CASE( "proper test","[2]" ) {
  vector<double> coefficients{3., 2.5, 2.1};
  REQUIRE_NOTHROW( polynomial(coefficients) );

  coefficients.at(0) = 0.;
  REQUIRE_THROWS( polynomial(coefficients) );
}
```

Write a constructor that accepts the coefficients, and throws an exception if the above condition is violated.

315. Odd degree polynomials only

With odd degree you can always find bounds x_-, x_+.
For this exercise we reject even degree polynomials.

```
// bisect/zeroclassmain.cpp
if ( not third_degree.is_odd() ) {
  cout << "This program only works for odd-degree polynomials\n";
  exit(1);
}
```

This test will be used later;
first we need to implement it.

Exercise 85: Odd degree testing

Implement the *is_odd* test.

Gain confidence by unit testing:

```
// bisect/testzeroarray.cpp
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) );
```

316. Test on polynomials evaluation

Next we need to evaluate polynomials.

Equality testing on floating point is dangerous:
use *Catch*::*Approx*(*sb*)

```
// bisect/zeroclasstest.cpp
polynomial second( {2,0,1.1} );
// correct interpretation: 2x^2 + 1.1
REQUIRE( second.evaluate_at(2) == Catch::Approx(9.1) );
// wrong interpretation: 1.1x^2 + 2
REQUIRE( second.evaluate_at(2) != Catch::Approx(6.4) );
polynomial third( {3,2,0,1} ); // 3x^3 + 2x^2 + 1
REQUIRE( third(0) == Catch::Approx(1) );
```

Exercise 86: Evaluation, looking neat

Make polynomial evaluation work, but use overloaded evaluation:

```
// bisect/zeroclasstest.cpp
polynomial second( {2,0,1.1} );
// correct interpretation: 2x^2 + 1.1
REQUIRE( second(2) == Catch::Approx(9.1) );
polynomial third( {3,2,0,1} ); // 3x^3 + 2x^2 + 1
REQUIRE( third(0) == Catch::Approx(1) );
```

317. Finding initial bounds

We need a function *find_initial_bounds* which computes x_-, x_+ such that

$$f(x_-) < 0 < f(x_+) \quad \text{or} \quad f(x_+) < 0 < f(x_-)$$

(can you write that more compactly?)

Exercise 87: Test for initial bounds

In the test for proper initial bounds, we reject even degree polynomials and left/right points that are reversed:

```
// bisect/zeroclasstest.cpp
double left{10},right{11};
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 );
double
  leftval = third(left),
  rightval = third(right);
REQUIRE( leftval*rightval<=0 );
```

Can you add a unit test on the left/right values?

318. Move the bounds closer

Root finding iteratively moves the initial bounds closer together:

```
// bisect/zeroclasslib.hpp
void move_bounds_closer
    ( const polynomial&,double& left,double& right,bool
    trace=false );
```

- on input, `left`$<$`right`, and
- on output the same must hold.
- ... but the bounds must be closer together.
- Also: catch various errors
- Also also: optional trace parameter; you leave that unusued.

Exercise 88: Test moving bounds

```
// bisect/zeroclasstest.cpp
  REQUIRE_THROWS( move_bounds_closer(third,right,left) );
  REQUIRE_THROWS( move_bounds_closer(third,left,left) );

  double old_left = left, old_right = right;
  REQUIRE_NOTHROW( move_bounds_closer(third,left,right) );
  leftval = third(left); rightval = third(right);
  REQUIRE( leftval*rightval<=0 );
  REQUIRE( ( ( left==old_left and right<old_right ) or
             ( right==old_right and left>old_left ) ) );
```

319. Putting it all together

Ultimately we need a top level function

```
double find_zero( polynomial coefficients,double prec );
```

- reject even degree polynomials
- set initial bounds
- move bounds closer until close enough:
  $|f(y)| < \texttt{prec}$.

Exercise 89: Put it all together

Make this call work:

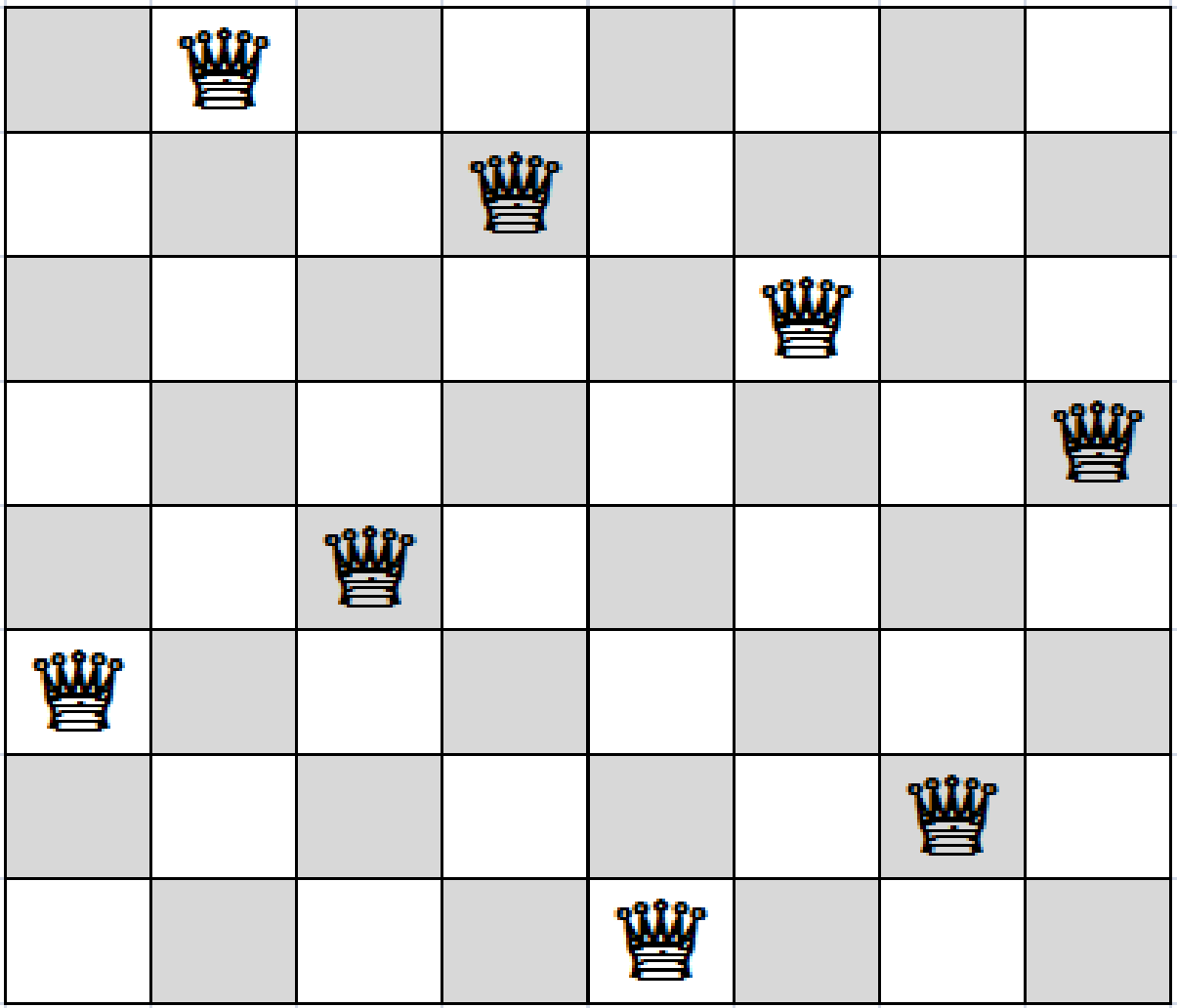
```
// bisect/zeroclassmain.cpp
auto zero = find_zero( coefficients, 1.e-8 );
cout << "Found root " << zero
     << " with value " << evaluate_at(coefficients,zero) << '\n';
```

Design unit tests, including on the precision attained, and make sure your code passes them.

TDD example: Eight queens

320. Classic problem

Can you put 8 queens on a board so that they can't hit each other?



321. Statement

- 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.

322. Not good solution

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.

Better: abort search early.

Exercise 90: Board class

Class *board*:

```
// queens/queens.hpp
ChessBoard(int n);
```

Method to keep track how far we are:

```
// queens/queens.hpp
int next_row_to_be_filled()
```

Test:

```
// queens/queentest.cpp
TEST_CASE( "empty board","[1]" ) {
  constexpr int n=10;
  ChessBoard empty(n);
  REQUIRE( empty.next_row_to_be_filled()==0 );
}
```

Exercise 91: Place one queen

Method to place the next queen,
without testing for feasibility:

```
// queens/queens.hpp
void place_next_queen_at_column(int i);
```

This test should catch incorrect indexing:

```
// queens/queentest.cpp
INFO( "Illegal placement throws" )
REQUIRE_THROWS( empty.place_next_queen_at_column(-1) );
REQUIRE_THROWS( empty.place_next_queen_at_column(n) );
INFO( "Correct placement succeeds" );
REQUIRE_NOTHROW( empty.place_next_queen_at_column(0) );
REQUIRE( empty.next_row_to_be_filled()==1 );
```

Without this test, would you be able to cheat?

Exercise 92: Test if we're still good

Feasibility test:

```
// queens/queens.hpp
bool feasible()
```

Some simple cases:
(add to previous test)

```
// queens/queentest.cpp
ChessBoard empty(n);
REQUIRE( empty.feasible() );

// queens/queentest.cpp
ChessBoard one = empty;
one.place_next_queen_at_column(0);
REQUIRE( one.next_row_to_be_filled()==1 );
REQUIRE( one.feasible() );
```

Exercise 93: Test collisions

```
// queens/queentest.cpp
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() );
```

Exercise 94: Test a full board

Construct full solution

```
// queens/queens.hpp
ChessBoard( int n,vector<int> cols );
ChessBoard( vector<int> cols );
```

Test:

```
// queens/queentest.cpp
ChessBoard five( {0,3,1,4,2} );
REQUIRE( five.feasible() );
```

Exercise 95: Exhaustive testing

This should now work:

```
// queens/queentest.cpp
// loop over all possibilities first queen
auto firstcol = GENERATE_COPY( range(1,n) );
ChessBoard place_one = empty;
REQUIRE_NOTHROW( place_one.place_next_queen_at_column(firstcol)
    );
REQUIRE( place_one.feasible() );

// loop over all possbilities second queen
auto secondcol = GENERATE_COPY( range(1,n) );
ChessBoard place_two = place_one;
REQUIRE_NOTHROW( place_two.place_next_queen_at_column(secondcol)
    );
if (secondcol<firstcol-1 or secondcol>firstcol+1) {
  REQUIRE( place_two.feasible() );
} else {
  REQUIRE( not place_two.feasible() );
}
```

Exercise 96: Place if possible

You need to write a recursive function:

```
// queens/queens.hpp
optional<ChessBoard> place_queens()
```

- place the next queen.
- if stuck, return 'nope'.
- if feasible, recurse.

```
class board {
  /* stuff */
  optional<board> place_queens() const {
    /* stuff */
    board next(*this);
    /* stuff */
    return next;
  };
```

Exercise 97: Test last step

Test *place_queens* on a board that is almost complete:

```
// queens/queentest.cpp
ChessBoard almost( 4, {1,3,0} );
auto solution = almost.place_queens();
REQUIRE( solution.has_value() );
REQUIRE( solution->filled() );
```

Note the new constructor! (Can you write a unit test for it?)

Exercise 98: Sanity tests

```
// queens/queentest.cpp
TEST_CASE( "no 2x2 solutions","[8]" ) {
  ChessBoard two(2);
  auto solution = two.place_queens();
  REQUIRE( not solution.has_value() );
}

// queens/queentest.cpp
TEST_CASE( "no 3x3 solutions","[9]" ) {
  ChessBoard three(3);
  auto solution = three.place_queens();
  REQUIRE( not solution.has_value() );
}
```

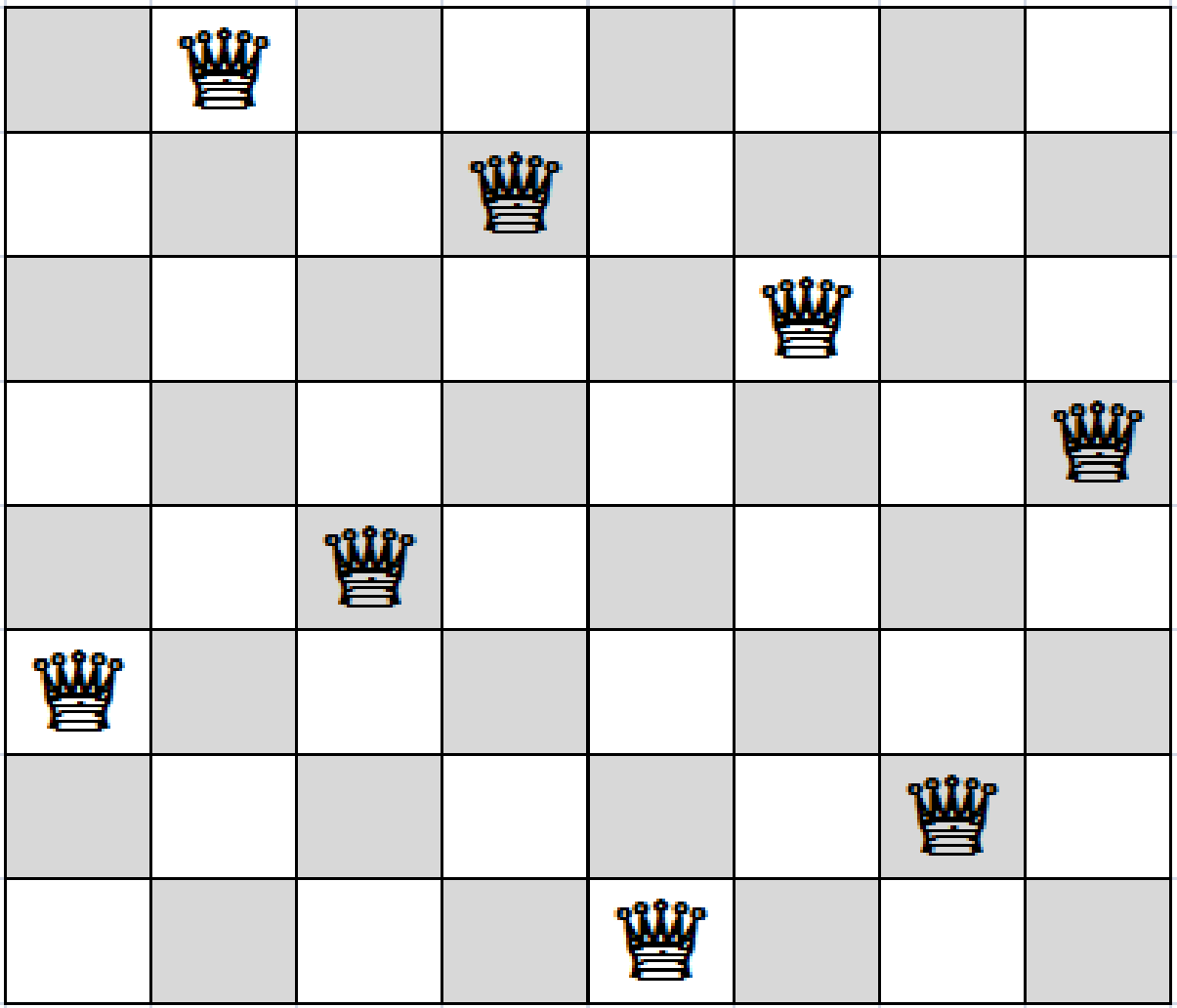
Exercise 99: O

ptional: can you do timing the solution time as function of the size of the board?

Eight queens problem by TDD (using objects)

323. Problem statement

Can you place eight queens on a chess board so that no pair threatens each other?

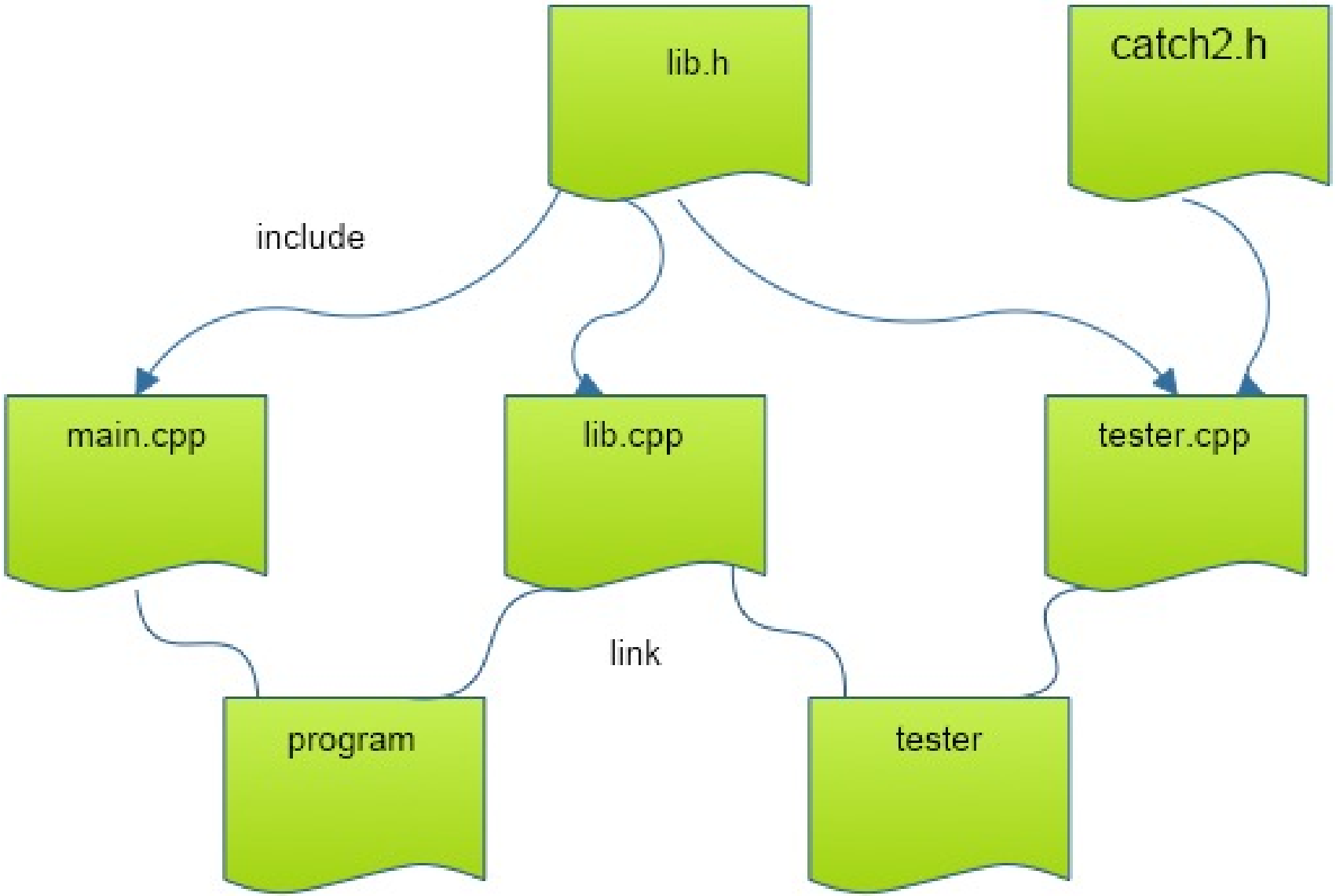


324. Sort of test-driven development

You will solve the ‘eight queens’ problem by

- designing tests for the functionality
- then implementing it

325. File structure



326. Basic object design

Object constructor of an empty board:

```
// queens/queens.hpp
ChessBoard(int n);
```

Test how far we are:

```
// queens/queens.hpp
int next_row_to_be_filled()
```

First test:

```
// queens/queentest.cpp
TEST_CASE( "empty board","[1]" ) {
  constexpr int n=10;
  ChessBoard empty(n);
  REQUIRE( empty.next_row_to_be_filled()==0 );
}
```

Exercise 100: Board object

Start writing the *board* class, and make it pass the above test.

Exercise 101: Board method

Write a method for placing a queen on the next row,

```
// queens/queens.hpp
void place_next_queen_at_column(int i);
```

and make it pass this test (put this in a *TEST_CASE*):

```
// queens/queentest.cpp
INFO( "Illegal placement throws" )
REQUIRE_THROWS( empty.place_next_queen_at_column(-1) );
REQUIRE_THROWS( empty.place_next_queen_at_column(n) );
INFO( "Correct placement succeeds" );
REQUIRE_NOTHROW( empty.place_next_queen_at_column(0) );
REQUIRE( empty.next_row_to_be_filled()==1 );
```

Exercise 102: Test for collisions

Write a method that tests if a board is collision-free:

```
// queens/queens.hpp
bool feasible()
```

This test has to work for simple cases to begin with. You can add these lines to the above tests:

```
// queens/queentest.cpp
ChessBoard empty(n);
REQUIRE( empty.feasible() );
```

```
// queens/queentest.cpp
ChessBoard one = empty;
one.place_next_queen_at_column(0);
REQUIRE( one.next_row_to_be_filled()==1 );
REQUIRE( one.feasible() );
```

```
// queens/queentest.cpp
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() );
```

Exercise 103: Test full solutions

Make a second constructor to 'create' solutions:

```
// queens/queens.hpp
ChessBoard( int n,vector<int> cols );
ChessBoard( vector<int> cols );
```

Now we test small solutions:

```
// queens/queentest.cpp
ChessBoard five( {0,3,1,4,2} );
REQUIRE( five.feasible() );
```

Exercise 104: No more delay: the hard stuff!

Write a function that takes a partial board, and places the next queen:

```
// queens/queens.hpp
optional<ChessBoard> place_queens()
```

Test that the last step works:

```
// queens/queentest.cpp
ChessBoard almost( 4, {1,3,0} );
auto solution = almost.place_queens();
REQUIRE( solution.has_value() );
REQUIRE( solution->filled() );
```

Alternative to using `optional`:

```
bool place_queen( const board& current, board &next );
// true if possible, false is not
```

Exercise 105: Test that you can find solutions

Test that there are no 3×3 solutions:

```
// queens/queentest.cpp
TEST_CASE( "no 3x3 solutions","[9]" ) {
  ChessBoard three(3);
  auto solution = three.place_queens();
  REQUIRE( not solution.has_value() );
}
```

but 4×4 solutions do exist:

```
// queens/queentest.cpp
TEST_CASE( "there are 4x4 solutions","[10]" ) {
  ChessBoard four(4);
  auto solution = four.place_queens();
  REQUIRE( solution.has_value() );
}
```

Turn it in!

- If you think your functions pass all tests, subject them to the tester:

  `coe_queens yourprogram.cc`

  where '`yourprogram.cc`' stands for the name of your source file.
- Is it reporting that your program is correct? If so, do:

  `coe_queens -s yourprogram.cc`

  where the `-s` flag stands for 'submit'.
- If you don't manage to get your code working correctly, you can submit as incomplete with

  `coe_queens -i yourprogram.cc`
- If you want feedback on what the tester thinks about your code do

  `coe_queens -d yourprogram.cc`

  with the `-d` flag for 'debug.

Dijkstra quote, part 1

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)

Dijkstra quote, part 2

The only effective way to raise the confidence level of a program significantly is to give a convincing proof of its correctness. But one should not first make the program and then prove its correctness, because then the requirement of providing the proof would only increase the poor programmer's burden. On the contrary: the programmer should let correctness proof and program grow hand in hand.

Matrix-vector product

$$y = Ax$$

Partitioned:

$$\begin{pmatrix} y_T \\ y_B \end{pmatrix} = \begin{pmatrix} A_T \\ A_B \end{pmatrix} (x)$$

Two equations:

$$\begin{cases} y_T = A_T x \\ y_B = A_B x \end{cases}$$

Inductive construction

$$\begin{pmatrix} y_T \\ y_B \end{pmatrix} = \begin{pmatrix} A_T \\ A_B \end{pmatrix} (x)$$

Assume only equation

$$y_T = A_T x$$

is satisfied, and grow the T block by one row.

Algorithm outline

$$\begin{pmatrix} y_T \\ y_B \end{pmatrix} = \begin{pmatrix} A_T \\ A_B \end{pmatrix} (x)$$

While T is not the whole system

Predicate: $y_T = A_T x$ true

Update: grow T block by one

Predicate: $y_T = A_T x$ true for new/bigger T block

Note initial and final condition.

Inductive step

Here is the big trick
Before

$$\begin{pmatrix} y_T \\ y_B \end{pmatrix} = \begin{pmatrix} A_T \\ A_B \end{pmatrix} (x)$$

split:

$$\begin{pmatrix} Y_1 \\ \cdots \\ y_2 \\ Y_3 \end{pmatrix} = \begin{pmatrix} A_1 \\ \cdots \\ a_2 \\ A_3 \end{pmatrix} (x)$$

Then the update step, and
After

$$\begin{pmatrix} Y_1 \\ y_2 \\ \cdots \\ Y_3 \end{pmatrix} = \begin{pmatrix} A_1 \\ a_2 \\ \cdots \\ A_3 \end{pmatrix} (x)$$

and unsplit

$$\begin{pmatrix} y_T \\ y_B \end{pmatrix} = \begin{pmatrix} A_T \\ A_B \end{pmatrix} (x)$$

Before the update:

$$\begin{pmatrix} Y_1 \\ \cdots \\ y_2 \\ Y_3 \end{pmatrix} = \begin{pmatrix} A_1 \\ \cdots \\ a_2 \\ A_3 \end{pmatrix} (x)$$

so

$$Y_1 = A_1 x$$

is true

Then the update step, and

After

$$\begin{pmatrix} Y_1 \\ y_2 \\ \cdots \\ Y_3 \end{pmatrix} = \begin{pmatrix} A_1 \\ a_2 \\ \cdots \\ A_3 \end{pmatrix} (x)$$

so

$$\begin{cases} Y_1 = A_1 x & \text{we had this} \\ y_2 = a_2 x & \text{we need this} \end{cases}$$

Resulting algorithm

While T is not the whole system

Predicate: $y_T = A_T x$ true

Update: $y_2 = a_2 x$

Predicate: $y_T = A_T x$ true for new/bigger T block

In traditional notation:

Do $n = 0..N-1$

Predicate: $y_{0:n} = A_{0:n,*} x$ true

Compute: $y_n = a_{n,*} x$

Predicate: $y_{0:n+1} = A_{0:n+1,*} x$ true

So what?

We already know this stuff, right?

Well, we made an assumption on how to partition the matrix.
What if we partition it differently?

Matrix-vector product, the other way around

$$y = Ax$$

Partitioned:

$$(y) = \begin{pmatrix} A_L & A_R \end{pmatrix} \begin{pmatrix} x_T \\ x_B \end{pmatrix}$$

Equation:

$$\left\{ y = A_L x_T + A_R x_B \right.$$

Inductive construction

$$(y) = \begin{pmatrix} A_L & A_R \end{pmatrix} \begin{pmatrix} x_T \\ x_B \end{pmatrix}$$

Assume

$$y = A_L x_T$$

is constructed, and grow the T block.

Inductive step

Before

$$(y) = \begin{pmatrix} A_L & A_R \end{pmatrix} \begin{pmatrix} x_T \\ x_B \end{pmatrix}$$

split:

$$(y) = \left(\begin{array}{c|cc} A_1 & A_2 & A_3 \end{array}\right) \begin{pmatrix} x_1 \\ \cdots \\ x_2 \\ x_3 \end{pmatrix}$$

Then the update step, and
After

$$(y) = \left(\begin{array}{cc|c} A_1 & A_2 & A_3 \end{array}\right) \begin{pmatrix} x_1 \\ x_2 \\ \cdots \\ x_3 \end{pmatrix}$$

and unsplit

$$(y) = \begin{pmatrix} A_L & A_R \end{pmatrix} \begin{pmatrix} x_T \\ x_B \end{pmatrix}$$

Derivation of the update

Before the update:

$$(y) = \begin{pmatrix} A_1 & \vdots & A_2 & A_3 \end{pmatrix} \begin{pmatrix} x_1 \\ \dots \\ x_2 \\ x_3 \end{pmatrix}$$

so

$$y = A_1 x_1$$

is true
Then the update step, and
After

$$(y) = \begin{pmatrix} A_1 & A_2 & \vdots & A_3 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \dots \\ x_3 \end{pmatrix}$$

so

$$y = A_1 x_1 + A_2 x_2$$

in other words, we need

$$y \leftarrow y + A_2 x_2$$

Resulting algorithm

While T is not the whole system

Predicate: $y = A_L x_T$ true

Update: $y \leftarrow y + A_2 x_2$

Predicate: $y = A_L x_T$ true for new/bigger T block

Two algorithms

for $r = 1, m$
    $y_r = A_{r,*} x_*$

$y \leftarrow 0$
for $c = 1, n$
    $y \leftarrow y + A_{*,c} x_c$

Take it from here

This gets more interesting if you take other algorithms:

- Matrix-matrix multiplication
- Matrix inversion
- Sylvester equation
- Iterative linear system solving

`https://shpc.oden.utexas.edu/`