C++ course

Victor Eijkhout, Susan Lindsey

COE 322 Fall 2021

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:

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

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

and compile it. Intel compiler:

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

Run this program (it gives no output):

```
./zeroprogram
```

2. 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:

```
cout << "Hello world!" << endl;
```

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

Compile and run again.

(Did you indent the 'hello world' line? Did your editor help you with the indentation?)

3. C++ versions

- The compiler by default uses `C++98`.
- This course explains `C++17`. 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'
  ```

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. Errors

There are two types of errors that your program can have:

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.

5. File names

File names can have extensions: the part after the dot.

- `program.cxx` or `program.cc` are typical extensions for C++ sources.
- `program.cpp` is also used, but there is a possible confusion with 'C PreProcessor'.
- Using `program` without extension usually indicates an executable.

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" << endl; // say hi!
```

and arbitrary:

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

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 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?

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. Typical 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.

11. Declaration

There are a couple of ways to make the connection between a name and a type. Here is a simple variable declaration, which establishes the name and the type:

```
int n;
float x;
int n1,n2;
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 that's not a good idea. Please only declare *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;
n1 = 7; n2 = n1 * 3;
```

These are not math equations: variable on the lhs gets the value of the rhs.

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

15. Special forms

Update:

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

Integer add/subtract one:

```
i++; j--;  /* same as: */ i=i+1; j=j-1;
```

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 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.

If you declare a variable but not initialize, you can not count on its value being anything, in particular not zero. Such implicit initialization is often omitted for performance reasons.

Review quiz 5

```
#include <iostream>
using std::cout;
using std::endl;
int main() {
  int i;
  int j = i+1;
  cout << j << endl;
  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.

(See Eijkhout, Introduction to High Performance Computing, chapter 3.)

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

You have already seen cout:

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

21. Terminal input

To make a program run dynamic, you can set starting values from keyboard input. For this, use `cin`, which takes keyboard input and puts it in a numerical (or string) variable.

```
// add at the top of your program:
using std::cin;

// then in your main:
int i;
cin >> i;
```

22. Quick intro to strings

- Add the following at the top of your file:

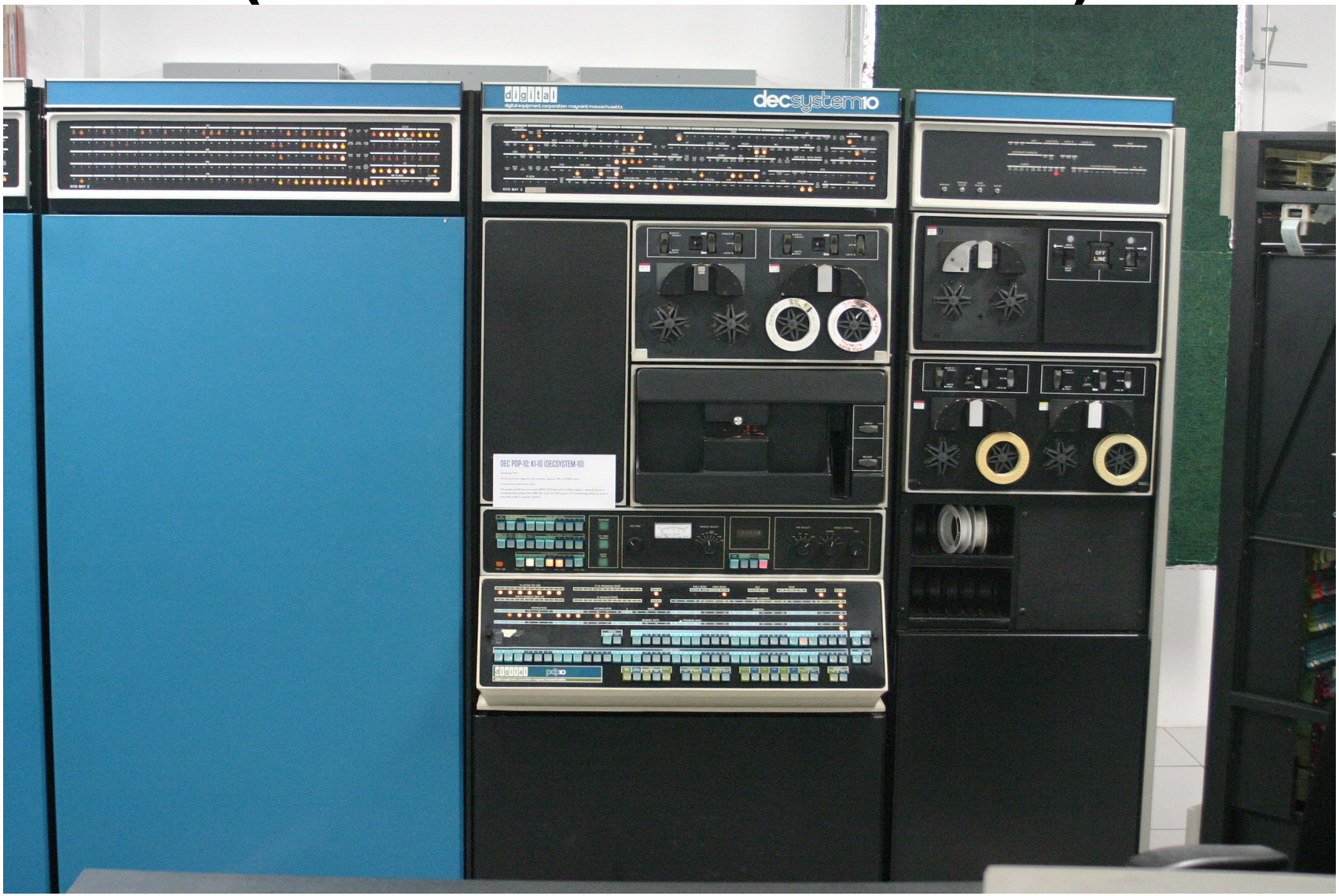
```
#include <string>
using std::string;
```

- 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 function 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 behaviour 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

Conditionals

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. What are logical expressions?

```
logical_expression ::
  comparison_expression
  | NOT comparison_expression
  | logical_expression CONJUNCTION comparison_expression
comparison_expression ::
  numerical_expression COMPARE numerical_expression
numerical_expression ::
  quantity
  | numerical_expression OPERATOR quantity
quantity :: number | variable
```

29. 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		`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:

```
int i;
cin >> i;
if ( i>100 )
  cout << "That number " << i << " had
      more than 2 digits" << endl;
```

Output

```
[basic] if:

... 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.cxx` 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" << endl;
else
  cout << "odd" << endl;
```

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 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.

Project Exercise 12

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

Code:

```
int number,divisor;
bool is_a_divisor;
/* ... */
if (
/* ... */
    ) {
  cout << "Indeed, " << divisor
       << " is a divisor of "
       << number << endl;
} else {
  cout << "No, " << divisor
       << " is not a divisor of "
       << number << endl;
}
```

Output

```
[primes] division:

( echo 6 ; echo 2 ) |
     1divisiontest
Enter a number:
Enter a trial divisor
    :
Indeed, 2 is a
    divisor of 6

( echo 9 ; echo 2 ) |
     1divisiontest
Enter a number:
Enter a trial divisor
    :
No, 2 is not a
    divisor of 9
```

30. Switch statement example

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

Code:

```
switch (n) {
case 1 :
case 2 :
  cout << "very small" << endl;
  break;
case 3 :
  cout << "trinity" << endl;
  break;
default :
  cout << "large" << endl;
}
```

Output

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

31. 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.

For loops

32. '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:

```
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 << endl;
```

33. 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!

34. Loop syntax: test

- If this test is true, do the next iteration.
- Done before the first iteration too!
- Test can be empty. What does this mean?

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

35. 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:

```
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 << endl;
```

and modify if 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*?

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.

36. 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.

37. 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 upperbound unspecified:

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

38. 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.

39. 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!

40. 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.

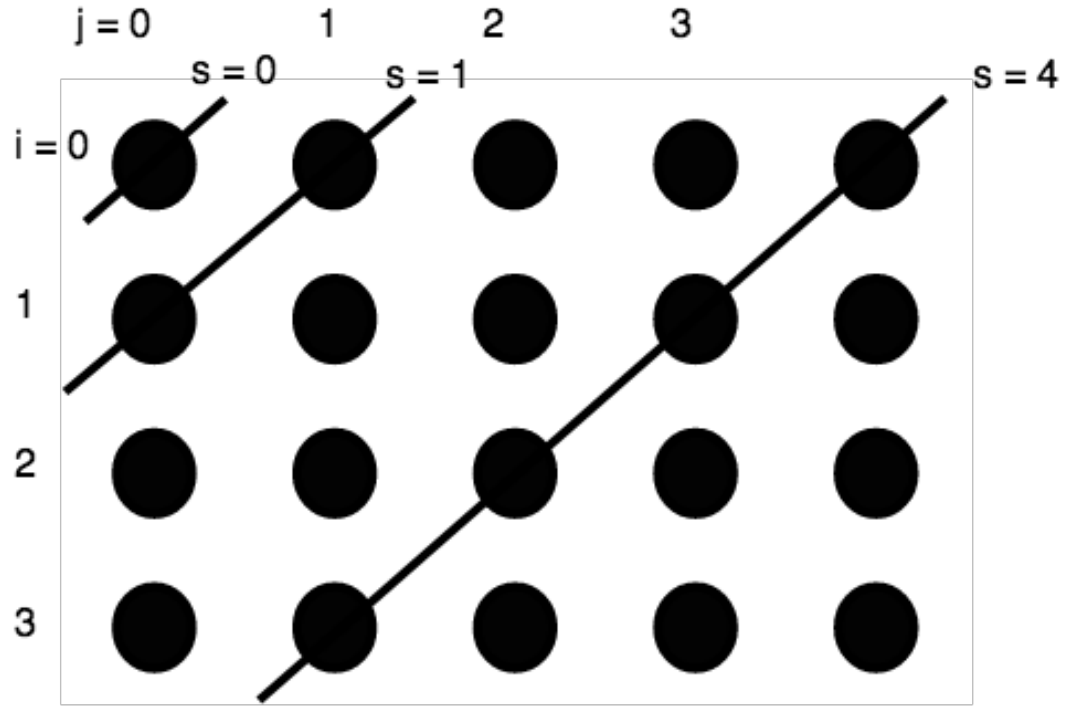
Exercise 18

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 $N = 40$.

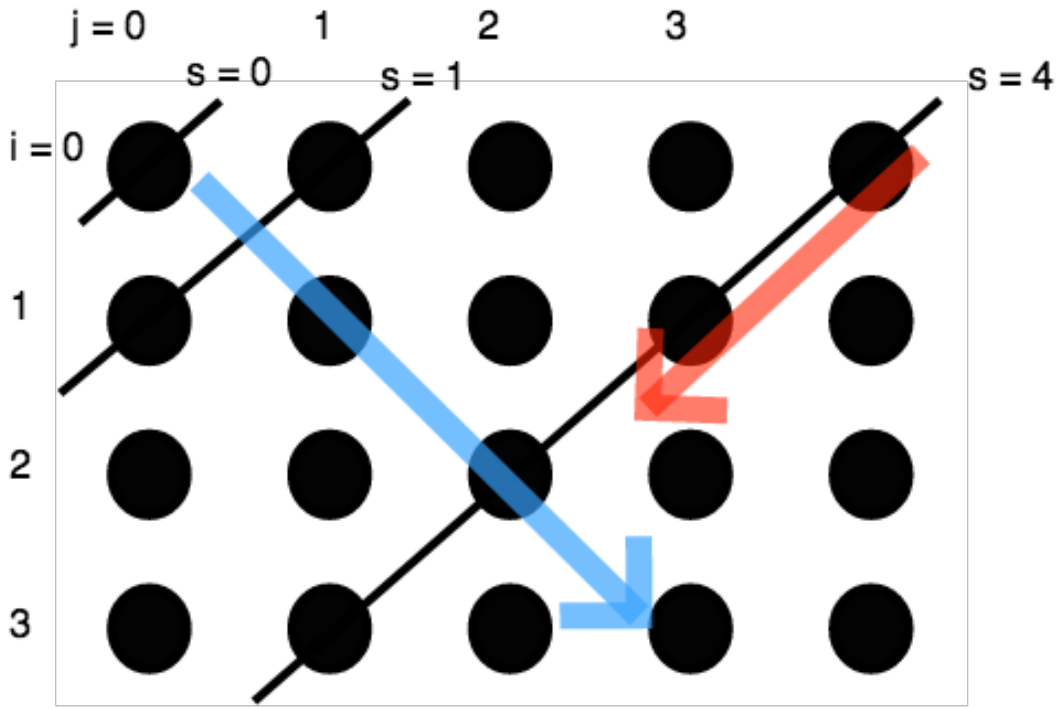
Secondly, find a pair with $i \cdot j > N$, but with the smallest value for $i + j$. (If there is more than one pair, report the one with lower i value.) Can you traverse the i, j indices such that they first enumerate all pairs $i + j = 1$, then $i + j = 2$, then $i + j = 3$ et cetera? Hint: write a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

41. 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

42. While loop

The other possibility for 'looping until' is a *while* loop, which repeats until a condition is met.

Syntax:

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

or

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

The while loop does not have a counter or an update statement; if you need those, you have to create them yourself.

43. Pre-test while loop

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

44. While syntax 1

Code:

```
cout << "Enter a positive number: " ;
cin >> invar; cout << endl;
cout << "You said: " << invar << endl;
while (invar<=0) {
  cout << "Enter a positive number: "
     ;
  cin >> invar; cout << endl;
  cout << "You said: " << invar <<
     endl;
}
cout << "Your positive number was "
     << invar << endl;
```

Output
[basic] whiledo:

```
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.

45. While syntax 2

Code:

```
int invar;
do {
  cout << "Enter a positive number: "
     ;
  cin >> invar; cout << endl;
  cout << "You said: " << invar <<
     endl;
} while (invar<=0);
cout << "Your positive number was: "
     << invar << endl;
```

Output

```
[basic] dowhile:

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

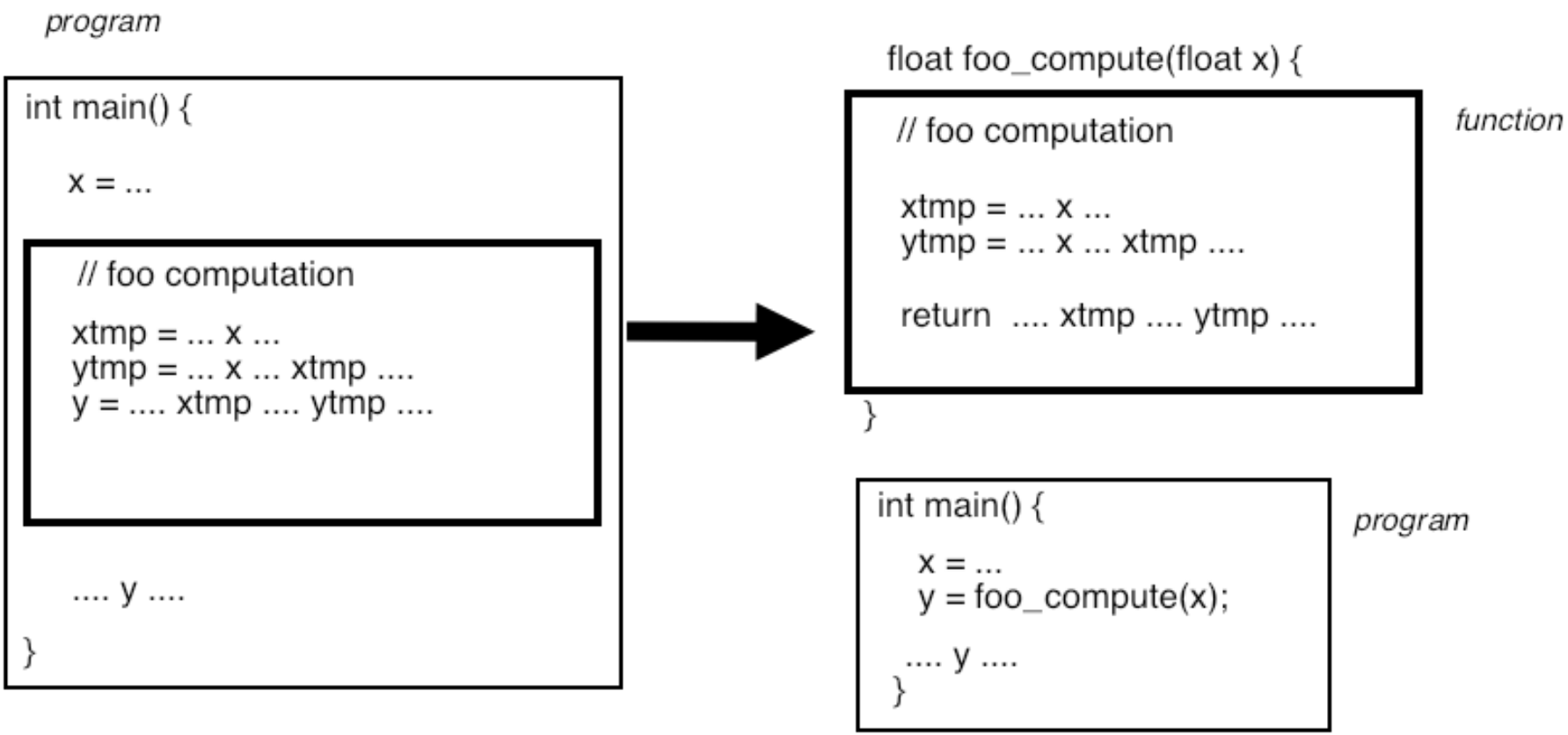
Function basics

46. 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.

47. Introducing a function



48. 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:

```
float left{0.},right{2.},
  mid;
while (right-left>.1) {
  mid = (left+right)/2.;
  float fmid =
    mid*mid*mid - mid*mid-1;
  if (fmid<0)
    left = mid;
  else
    right = mid;
}
cout << "Zero happens at: " << mid <<
    endl;
```

Output

```
[func] bisect1:

Zero happens at:
     1.4375
```

49. Introducing functions, step1

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

```
float f(float x) {
  return x*x*x - x*x-1;
};
```

Used in main:

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

50. Introducing functions, step 2

Add function for zero finding:

(note the local variable)

```
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;
};
```

The main no longer contains any implementation details (local variables, method used):

```
int main() {
  float left{0.},right{2.};
  float zero =
    find_zero_between(left,
    right);
  cout << "Zero happens at: "
       << zero << endl;
  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.cxx` in the repository

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

51. 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

52. 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);
```

53. 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 defitions can go in or out main" "T" "F"`

54. Prototype first, definition last

```
int my_computation(int);
int main() {
  int result;
  result = my_computation(5);
  return 0;
};
int my_computation(int i) {
   return i+3;
}
```

55. 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.

56. 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: $1, 0$; $0, 1$; $1, 1$; $3, 4$.

Project Exercise 22

Write a function `test_if_prime` that has an integer parameter, and returns a boolean corresponding to whether the parameter was prime.

```
int main() {
  bool isprime;
  isprime = test_if_prime(13);
```

Read the number in, and print the value of the boolean.

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?

Project Exercise 23

Take your prime number testing function `test_if_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'.)

57. Background: square roots by Newton's

Early computers had no hardware for computing a square root. Instead, they used Newton's method. Suppose you have a value y and you want want to compute $x = \sqrt{y}$. 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

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

Parameter passing

58. 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'

59. Pass by value example

Note that the function alters its parameter `x`:

Code:

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

Output

```
[func] passvalue:

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.

60. Reference

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

Code:

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

Output
[basic] ref:

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

(You will not use references often this way.)

61. Parameter passing by reference

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

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

62. 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'.

63. Pass by reference example 1

Code:

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

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

Output

```
[basic] setbyref:

5
```

Compare the difference with leaving out the reference.

64. 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' ....
```

However, see `std::optional`; section **??**.

Exercise 25

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

Code:

```
cout << i << "," << j << endl;
swap(i,j);
cout << i << "," << j << endl;
```

Output

```
[func] swap:

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:

```
cout << number;
if (is_divisible(number,divisor,
     remainder))
  cout << " is divisible by ";
else
  cout << " has remainder "
       << remainder << " from ";
cout << divisor << endl;
```

Output

```
[func] divisible:

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:

```
const float pi = 2*acos(0.0);
float x{1.}, y{0.};
rotate(x,y,pi/4);
cout << "Rotated halfway: ("
     << x << "," << y << ")" << endl;
rotate(x,y,pi/4);
cout << "Rotated to the y-axis: ("
     << x << "," << y << ")" << endl;
```

Output

```
[geom] rotate:

Rotated halfway:
     (0.707107,0.707107)

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

Recursion

65. 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:

```
int times( int number,int mult ) {
  cout << "(" << mult << ")";
  if (mult==1)
    return number;
  else
    return number + times(number,mult
     -1);
}
```

Output

```
[func] mult:

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

Extend this idea to define powers as repeated multplication.

You can base this off the file `mult.cxx` 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:
  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

66. 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.

67. 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
```

68. 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

69. 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
}
```

70. 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?

```
bool something{false};
int i = 3;
if ( something ) {
  int i = 5;
  cout << "Local: " << i << endl;
}
cout << "Global: " << i << endl;
if ( something ) {
  float i = 1.2;
  cout << i << endl;
  cout << "Local again: " << i << endl;
}
cout << "Global again: " << i << endl;
```

71. Life time vs reachability

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

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

Arrays

Vectors

72. What are vectors?

- Linear storage of items:
  (often called 'array' but that has other meanings)
  - items of any type (but the same for all elements of one vector)
  - potentially very many items
- Indexed set of items
- . . . but if you don't need the index: collection of items

73. Short vectors

Short vectors can be created by enumerating their elements:

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

int main() {
  vector<int> evens{0,2,4,6,8};
  vector<float> halves = {0.5, 1.5, 2.5};
  cout << evens.at(0) << endl;
  return 0;
}
```

Exercise 33

1. Take the above snippet, supply the missing header lines, 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, containing odd numbers, which are the even values plus 1?

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

74. Range over elements

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

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

Code:

```
vector<int> numbers = {1,4,2,6,5};
int tmp_max = -2000000000;
for (auto v : numbers)
  if (v>tmp_max)
    tmp_max = v;
cout << "Max: " << tmp_max
     << " (should be 6)" << endl;
```

Output

```
[array] dynamicmax:

Max: 6 (should be 6)
```

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);
```

75. Range over vector denotation

Code:

```
for ( auto i : {2,3,5,7,9} )
  cout << i << ",";
cout << endl;
```

Output

```
[array] rangedenote:

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

76. Vector definition

Definition and/or initialization:

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

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

vector<int> numbers = {1,4};
numbers[0] += 3;
numbers[1] = 8;
cout << numbers[0] << ","
     << numbers[1] << endl;
```

```
Output
[array] assignbracket:

4,8
```

With bound checking:

```
Code:

vector<int> numbers = {1,4};
numbers.at(0) += 3;
numbers.at(1) = 8;
cout << numbers.at(0) << ","
     << numbers.at(1) << endl;
```

```
Output
[array] assignatfun:

4,8
```

Safer, slower.

(Remember Knuth about optimization.)

78. Vector elements out of bounds

Square bracket notation:

```
Code:

vector<int> numbers = {1,4};
numbers[-1] += 3;
numbers[2] = 8;
cout << numbers[0] << ","
     << numbers[1] << endl;
```

```
Output
[array]
assignoutofboundbracket:

1,4
```

With bound checking:

```
Code:

vector<int> numbers = {1,4};
numbers.at(-1) += 3;
numbers.at(2) = 8;
cout << numbers.at(0) << ","
     << numbers.at(1) << endl;
```

```
Output
[array]
assignoutofboundatfun:

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

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:

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

Output

```
[array] vectorrangeref:

6.6
```

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

80. 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 array, and where it occurs.

Code:

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

Output

```
[array] vecidxmax:

Max: 6.6 at index: 3
```

81. 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 35

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

Which mechanism do you use?

Exercise 36

Create a `vector` x of `float` elements, and set them to random values.

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?

82. 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);
```

83. Vector copy

Vectors can be copied just like other datatypes:

Code:

```
vector<float> v(5,0), vcopy;
v.at(2) = 3.5;
vcopy = v;
vcopy.at(2) *= 2;
cout << v.at(2) << ","
     << vcopy.at(2) << endl;
```

Output

```
[array] vectorcopy:

3.5,7
```

84. 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`.

85. 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. More later.

Dynamic behaviour

86. Dynamic vector extension

Extend a vector's size with `push_back`:

Code:

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

Output

```
[array] vectorend:

6
35
```

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

87. 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)`.

88. Filling in vector elements

You can push elements into a vector:

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

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

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

89. Filling in vector elements

With subscript:

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

You can also use `new` to allocate*:

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

*Considered bad practice. Do not use.

90. 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

91. 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.

> *C difference: In C the array is not copied: you pass the address by value. Not the contents.*

92. Vector pass by value example

Code:

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

Output

```
[array] vectorpassnot:

3.5
```

93. Vector pass by reference

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

Code:

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

Output

```
[array] vectorpassref:

4.6
```

Exercise 37

Revisit exercise 36 and introduce a function for computing the L_2 norm.

94. 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:

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

Output

```
[array] vectorreturn:

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

Exercise 38

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

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

Can you write a function that accepts a vector and produces two vectors as described?

(hints for the next exercise)

```
// 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
```

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

95. 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.

```
class named_field {
private:
  vector<double> values;
  string name;
```

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

96. Create the contained vector

Use initializers for creating the contained vector:

```
class named_field {
private:
  vector<int> values;
public:
  named_field( int n )
    : values(vector<int>(n)) {
  };
};
```

Less desirable method is creating in the constructor:

```
  named_field( int n ) {
    values = vector<int>(n);
  };
```

Multi-dimensional arrays

97. Multi-dimensional vectors

Multi-dimensional is harder with vectors:

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

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

98. Matrix class

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

Exercise 40

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

99. Matrix class; better design

Better idea:

```
class Matrix {
private:
  int rows,cols;
  vector<double> elements;
private:
  Matrix( int m,int n )
  : rows(m),cols(n),
    elements = vector<double>(rows*cols)
    {};
  ...
  double get(int i,int j) {
    return elements.at(i*cols+j);
  }
```

(Old-style solution: use cpp macro)

Exercise 41

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

Exercise 42

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

Implement matrix-matrix multiplication.

100. 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 43

- 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.

```
           *
          * *
         *   *
        * * * *
       *       *
      * *     * *
     *   *   *   *
    * * * * * * * *
   *               *
  * *             * *
```

The object needs to have an array internally. The easiest solution is

Optional exercise 44

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
  ```

Static arrays

Array creation

C-style arrays still exist,

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

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

Ranging

You can range over static arrays same as for `vector`

I/O

Formatted output

101. Formatted output

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

```
#include <iomanip>
```

102. Default unformatted output

Code:

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

Output

```
[io] cunformat:

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

103. Reserve space

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

Code:

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

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

Output

```
[io] width:

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

Width is 6:
>     123
```

104. Padding character

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

Code:

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

Output
[io] formatpad:

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

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

105. Left alignment

Instead of right alignment you can do left:

Code:

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

Output
[io] formatleft:

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

106. Number base

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

Code:

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

Output

```
[io] format16:

0 1 2 3 4 5 6 7 8 9 a
   b c d e f
10 11 12 13 14 15 16
  17 18 19 1a 1b 1c
   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
```

Exercise 45

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
```

Exercise 46

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

Code:

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

Output

```
[io] quasifix:

   1.5
  12.32
 123.456
1234.5678
```

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

107. Hexadecimal

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

Code:

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

Output

```
[pointer] coutpoint:

address of i, decimal
    : 140732704635892
address of i, hex
    : 0x7ffee2de3bf4
```

Back to decimal:

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

Floating point formatting

108. Floating point precision

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

Code:

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

Output

```
[io] formatfloat:

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

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

109. Fixed point precision

Fixed precision applies to fractional part:

Code:

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

Output

```
[io] fix:

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

(Notice the rounding)

110. Aligned fixed point output

Combine width and precision:

Code:

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

Output
[io] align:

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

111. Scientific notation

Combining width and precision:

Code:

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

Output

```
[io] iofsci:

1.2346e+00
1.2346e+01
1.2346e+02
1.2346e+03
1.2346e+04
1.2346e+05
1.2346e+06
1.2346e+07
1.2346e+08
1.2346e+09
```

File output

112. Text output to file

The *iostream* is just one example of a stream:
general mechanism for converting entities to exportable form.
In particular: file output works the same as screen output.

Use:

Code:

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

Output
[io] fio:

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

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

113. Binary output

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

Code:

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

Output
[io] fiobin:

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

Cout on classes (for future reference)

114. 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.

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

Strings

Characters

115. Characters and ints

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

116. Char / int equivalence

Equivalent to (short) integer:

Code:

```
char ex = 'x';
int x_num = ex, y_num = ex+1;
char why = y_num;
cout << "x is at position " << x_num
     << endl;
cout << "one further lies " << why
     << endl;
```

Output
[string] intchar:

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

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

Exercise 47

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

117. String declaration

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

// .. and now you can use ‘string’
```

(Do not use the C legacy mechanisms.)

118. 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";
```

119. Quotes in strings

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

Code:

```
string
  one("a b c"),
  two("a \"b\" c"),
  three( R"("a ""b """c)" );
cout << one << endl;
cout << two << endl;
cout << three << endl;
```

Output
[string] quote:

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

120. Concatenation

Strings can be *concatenated*:

Code:

```
string my_string, space{" "};
my_string = "foo";
my_string += space + "bar";
cout << my_string << ": " << my_string
     .size() << endl;
```

Output

```
[string] stringadd:

foo bar: 7
```

121. String indexing

You can query the *size*:

Code:

```
string five_text{"fiver"};
cout << five_text.size() << endl;
```

Output
[string] stringsize:

5

or use subscripts:

Code:

```
string digits{"0123456789"};
cout << "char three: "
     << digits[2] << endl;
cout << "char four : "
     << digits.at(3) << endl;
```

Output
[string] stringsub:

char three: 2
char four : 3

122. Ranging over a string

Same as ranging over vectors.

Range-based for:

Code:

```
cout << "By character: ";
for ( char c : abc )
  cout << c << " ";
cout << endl;
```

Output

```
[string] stringrange:

By character: a b c
```

Ranging by index:

Code:

```
string abc = "abc";
cout << "By character: ";
for (int ic=0; ic<abc.size(); ic++)
  cout << abc[ic] << " ";
cout << endl;
```

Output

```
[string] stringindex:

By character: a b c
```

123. Range with reference

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

Code:

```
for ( char &c : abc )
  c += 1;
cout << "Shifted: " << abc << endl;
```

Output

```
[string] stringrangeset:

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 48

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

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

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

124. More vector methods

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

Code:

```
string five_chars;
cout << five_chars.size() << endl;
for (int i=0; i<5; i++)
  five_chars.push_back(' ');
cout << five_chars.size() << endl;
```

Output
[string] stringpush:

```
0
5
```

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

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

Exercise 49

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 50

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

125. 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.

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

Exercise 51

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

Code:

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

Output

```
[io] quasifix:

   1.5
  12.32
 123.456
1234.5678
```

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

Objects

Classes

126. Definition of object

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'.

Objects come in classes. A class is like a datatype: you can make objects of a class like variables of a datatype.

127. Running example

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

```
class Point {
    /* stuff */
};
int main () {
  Point p; /* stuff */
}
```

Exercise 52

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

128. Object functionality

Small illustration: vector objects.

Code:

```
Point p(1.,2.); // make point (1,2)
cout << "distance to origin "
     << p.distance_to_origin() << endl
     ;
p.scaleby(2.);
cout << "distance to origin "
     << p.distance_to_origin() << endl
     << "and angle " << p.angle()
     << endl;
```

Output
[object] functionality:

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

Note the 'dot' notation. Pronounce with 'apostrophe-ess': 'p's distance' et cetera.

Exercise 53

Thought exercise:
What data does the object need to store to do this?
Is there more than one possibility?

129. The object workflow

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

```
class myobject { /* ... */ };
```

  (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( /* ... */ );
```

130. Constructor and data initialization

To create an object belonging to a class use a constructor: function with same name as the class.

Constructors are typically used to initialize data members.

```
class Point {
private: // members
  double x,y;
public: // methods
  Point( double in_x,
         double in_y ) {
    x = in_x; y = in_y;
  };
};
```

```
Point v(1.,2.);
```

Methods

131. Class methods

Let's define method *distance*.

Definition in the class:

```
class Point {
  /* stuff */
  double distance_to_origin() {
    return sqrt(x*x + y*y); };
}
```

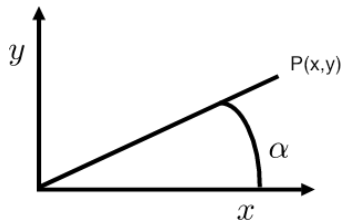
Use in the program:

```
Point pt(5,12);
double
  s = pt.distance_to_origin();
```

- 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 54

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.cxx` *in the repository*

Exercise 55

Make a class *GridPoint* which can 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.

132. Member default values

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

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

Each object will have its members initialized to these values.

133. Food for thought: initialization vs stora

The members stored can be different from the constructor arguments.

Example: create a vector from `x,y` cartesian coordinates, but store `r,theta` polar coordinates:

```
#include <cmath>
class Point {
private: // members
  double r,theta;
public: // methods
  Point( double x,double y ) {
    r = sqrt(x*x+y*y);
    theta = atan2(y/x);
  }
```

Exercise 56

Discuss the pros and cons of this design:

```
class Point {
private:
  double x,y,alpha;
public:
  Point(double x,double y)
  : x(x),y(y) {
    alpha = // something trig
  };
  double angle() { return alpha; };
};
```

134. Data access in methods

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

```
class Point {
private:
  double x,y;
public:
 void flip() {
    Point flipped;
    flipped.x = y; flipped.y = x;
    // more
  };
};
```

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

Exercise 57

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.

Hint: remember the 'dot' notation for members.

Review quiz 14

T/F?

- A class is primately determined by the data it stores.
  `/poll "Class determined by its data" "T" "F"`
- A class is primarily determing 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"`

135. Methods that alter the object

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

Code:

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

Output

```
[geom] pointscaleby:

p1 to origin 2.23607
p1 to origin 4.47214
```

Interaction between objects

136. Methods that create a new object

Code:

```
class Point {
  /* ... */
  Point scale( double a );
  /* ... */
  cout << "p1 to origin "
       << p1.length() << endl;
  Point p2 = p1.scale(2.);
  cout << "p2 to origin "
       << p2.length() << endl;
```

Output

```
[geom] pointscale:
p1 to origin 2.23607
p2 to origin 4.47214
```

137. Anonymous objects

Two ways of returning the scaled point:

Naive:

```
Point Point::scale( double a )
    {
  Point scaledpoint =
    Point( x*a, y*a );
  return scaledpoint;
};
```

No copy involved:

```
Point Point::scale( double a )
    {
  return Point( x*a, y*a );
};
```

Optional exercise 58

Write a method `halfway_point` 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?

138. Default constructor

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 (g++; different for intel):

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

139. Default constructor

The problem is with *p2*:

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

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

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

- as soon as you define a constructor, the default constructor goes away;
- you need to redefine the default constructor:

  ```
  Point() {};
  Point( double x,double y )
    : x(x),y(y) {};
  ```

140. Public versus private

- Interface: `public` functions that determine the functionality of the object; effect on data members is secondary.
- Implementation: data members, keep `private`: they only support the functionality.

This separation is a Good Thing:

- Protect yourself against inadvertant changes of object data.
- Possible to change implementation without rewriting calling code.

Exercise 59

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;
```

141. Classes for abstract objects

Objects can model fairly abstract things:

Code:

```
class Stream {
private:
  int last_result{0};
public:
  int next() {
    return last_result++; };
};

int main() {
  Stream ints;
  cout << "Next: "
       << ints.next() << endl;
  cout << "Next: "
       << ints.next() << endl;
  cout << "Next: "
       << ints.next() << endl;
```

Output

```
[object] stream:

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

Project Exercise 60

Write a class `primegenerator` that contains

- members `how_many_primes_found` and `last_number_tested`,
- a method `nextprime`;
- Also write a function `isprime` that does not need to be in the class.

Your main program should look as follows:

```
cin >> nprimes;
primegenerator sequence;
while (sequence.number_of_primes_found()<nprimes) {
  int number = sequence.nextprime();
  cout << "Number " << number << " is prime" << endl;
}
```

Project Exercise 61

The Goldbach conjecture says that every even number, from 4 on, is the sum of two primes $p + q$. Write a program to test this for the even numbers up to a bound that you read in. Use the `primegenerator` class you developed in exercise 264.

This is a great exercise for a top-down approach!

1. Make an outer loop over the even numbers e.
2. For each e, generate all primes p.
3. From $p + q = e$, it follows that $q = e - p$ is prime: test if that q is prime.

For each even number `e` then print `e,p,q`, for instance:

```
The number 10 is 3+7
```

If multiple possibilities exist, only print the first one you find.

Turn it in!

- If you have compiled your program, do:

  `coe_goldbach 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_goldbach -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_goldbach -i yourprogram.cc`

Other object stuff

142. String an object

1. Define a function that yields a string representing the object, and
2. redefine the less-less operator to use this.

```
#include <sstream>
using std::stringstream;
#include <string>
using std::string;
  /* ... */
  string as_string() {
    stringstream ss;
    ss << "(" << x << "," << y
    << ")";
    return ss.str();
  };
  /* ... */
```

```
std::ostream& operator<<
    (std::ostream &out,Point &p
    ) {
  out << p.as_string(); return
    out;
};
  /* ... */
    Point p1(1.,2.);
    cout << "p1 " << p1
         << " has length "
         << p1.length() << endl
    ;
```

143. Class prototypes

Header file:

```
class something {
private:
  int i;
public:
  double dosomething( int i, char c );
};
```

Implementation file:

```
double something::dosomething( int i, char c ) {
  // do something with i,c
};
```

Advanced stuff about constructors

144. Copy constructor

- Default defined copy and 'copy assignment' constructors:

```
some_object x(data);
some_object y = x;
some_object z(x);
```

- They copy an object:
  - simple data, including pointers
  - included objects recursively.
- You can redefine them as needed.

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

145. Copy constructor in action

Code:

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

Output

```
[object] copyscalar:

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

146. Copying is recursive

Class with a vector:

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

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

Code:

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

Output

```
[object] copyvector:

I have: 3
I have: 5
```

147. Destructor

- Every class `myclass` has a *destructor* `~myclass` defined by default.
- The default destructor does nothing:

  ```
  ~myclass() {};
  ```

- A destructor is called when the object goes out of scope. Great way to prevent memory leaks: dynamic data can be released in the destructor. Also: closing files.

148. Destructor example

Just for tracing, constructor and destructor do `cout`:

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

149. Destructor example

Destructor called implicitly:

Code:

```
cout << "Before the nested scope"
     << endl;
{
  SomeObject obj;
  cout << "Inside the nested scope"
       << endl;
}
cout << "After the nested scope"
     << endl;
```

Output

```
[object] destructor:

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

Hierarchical object relations

- You have seen inclusion relations.
- Hierarchical: object belongs to class, but als broader class
- 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.

Terminology

- 'Polygon' is the *base class*.
- 'Triangle' is a *derived class*.
- Derived classes *inherit* data and methods from the base class.

150. Examples for base and derived cases

- Base case: employee. Has: salary, employee number.
  Special case: manager. Has in addition: underlings.
- Base case: shape in drawing program. Has: extent, area, drawing routine.
  Special case: square et cetera; has specific drawing routine.

151. 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

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

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

152. Inheritance: derived classes

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

```
int main() {
  Special special_object;
  special_object.general_method();
  special_object.special_method();
}
```

Members and methods need to be `protected`, not `private`, to be inheritable.

153. Constructors

When you run the special case constructor, usually the general constructor needs to run too. By default the 'default constructor', but usually explicitly invoked:

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

154. 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 62

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 63

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.

155. Overriding methods

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

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

156. 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'.

157. Has-a relationship

A class usually contains data members. These can be simple types or other classes. This allows you to make structured code.

```
class Course {
private:
  Person the_instructor;
  int year;
}
class Person {
  string name;
  ....
}
```

This is called the has-a relation.

158. Literal and figurative has-a

A line segment has a starting point and an end point.

A Segment class can store those points:

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

or store one and derive the other:

```
class Segment {
private:
  Point starting_point;
  float length,angle;
public:
  Point get_the_end_point() {
    /* some computation
       from the
       starting point */ };
}
```

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

159. Default Constructors in initialization

```
class Inner { /* ... */ };
class Outer {
private:
  Inner inside_thing;
```

Two possibilities for constructor:

```
Outer( Inner thing )
: inside_thing(thing) {};
```

The *Inner* object is copied during construction of *Outer* object.

```
Outer( Inner thing ) {
  inside_thing = thing;
};
```

The *Outer* object is created, including construction of *Inner* object, then the argument is copied into place: $\Rightarrow$ needs default constructor on *Inner*.

Exercise 64

1. Make a class `Rectangle` (sides parallel to axes) with a constructor:

   ```
   Rectangle(Point bl,float w,float h);
   ```

   The logical implementation is to store these quantities. Implement methods

   ```
   float area(); float rightedge(); float topedge();
   and write a main program to test these.
   ```

2. Add a second constructor

   ```
   Rectangle(Point bl,Point tr);
   ```

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

3. Make a copy of your file, and redesign your class so that it stores two `Point` objects. Your main program should not change.

160. Polymorphism in constructors

You have to decide what to store and what to derive, but you can construct two ways:

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

Advantage: with a good API you can change your mind about the implementation without changing the calling code.

Pointers

161. Recursive data structures

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

This does not work: would take infinite memory.

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

PointToNode 'points' to the first node of the tail.

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.

162. Simple example

Simple class that stores one number:

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

163. Creating a shared pointer

Allocation and pointer in one:

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

164. Headers for smart pointers

Using shared pointers requires at the top of your file:

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

165. What's the point of pointers?

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

Code:

```
auto xptr = make_shared<HasX>(5);
auto yptr = xptr;
cout << xptr->get() << endl;
yptr->set(6);
cout << xptr->get() << endl;
```

Output

```
[pointer] twopoint:

5
6
```

Automatic memory management

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
}
// the variable 'array' does not exist anymore
// but the memory is still reserved.
```

The application 'is leaking memory'.
Java/Python have 'garbage collection': runtime impact
C++ has the best solution: smart pointers.

166. Reference counting illustrated

We need a class with constructor and destructor tracing:

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

167. Pointer overwrite

Let's create a pointer and overwrite it:

Code:

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

Output

```
[pointer] ptr1:

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

168. Pointer copy

Code:

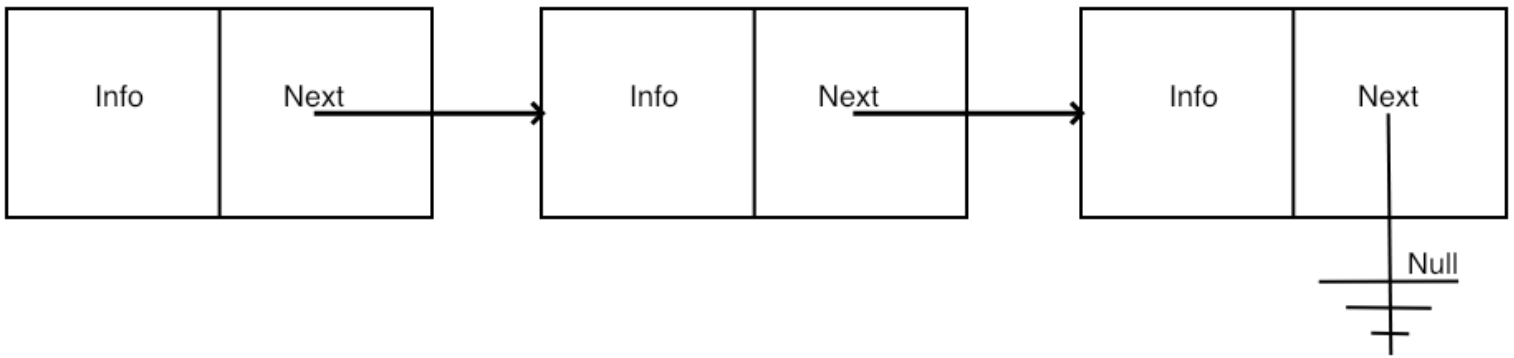
```
cout << "set pointer2" << endl;
auto thing_ptr2 =
  make_shared<thing>();
cout << "set pointer3 by copy"
     << endl;
auto thing_ptr3 = thing_ptr2;
cout << "overwrite pointer2"
     << endl;
thing_ptr2 = nullptr;
cout << "overwrite pointer3"
     << endl;
thing_ptr3 = nullptr;
```

Output

```
[pointer] ptr2:
set pointer2
.. calling
    constructor
set pointer3 by copy
overwrite pointer2
overwrite pointer3
.. calling destructor
```

Example: linked lists

Linked list



169. Linked lists

The prototypical example use of pointers is in linked lists. Consider a class *Node* with

- a data value to store, and
- a pointer to another *Node*, or `nullptr` if none.

Constructor sets the data value:

```
class Node {
private:
  int datavalue{0};
  shared_ptr<Node>
    tail_ptr{nullptr};
public:
  Node() {}
  Node(int value)
    : datavalue(value) {};
  int value() { return
    datavalue; };
```

Set next / test if there is a next:

```
bool has_next() {
  return tail_ptr!=nullptr; };
```

170. List usage

Example use:

Code:

```
auto
  first = make_shared<Node>(23),
  second = make_shared<Node>(45);
first->set_tail(second);
cout << "List length: "
     << first->list_length() << endl;
first->print();
```

Output

```
[tree] simple:

List <<23,45>> has
     length 2
```

171. Linked lists and recursion

Many operations on linked lists can be done recursively:

```
int Node::list_length() {
  if (!has_next()) return 1;
  else return 1+tail_ptr->list_length();
};
```

Exercise 65

Write a method *set_tail* that sets the tail of a node.

```
Node one;
one.set_tail( two ); // what is the type of 'two'?
cout << one.list_length() << endl; // prints 2
```

Exercise 66

Write a recursive *append* method that appends a node to the end of a list:

Code:

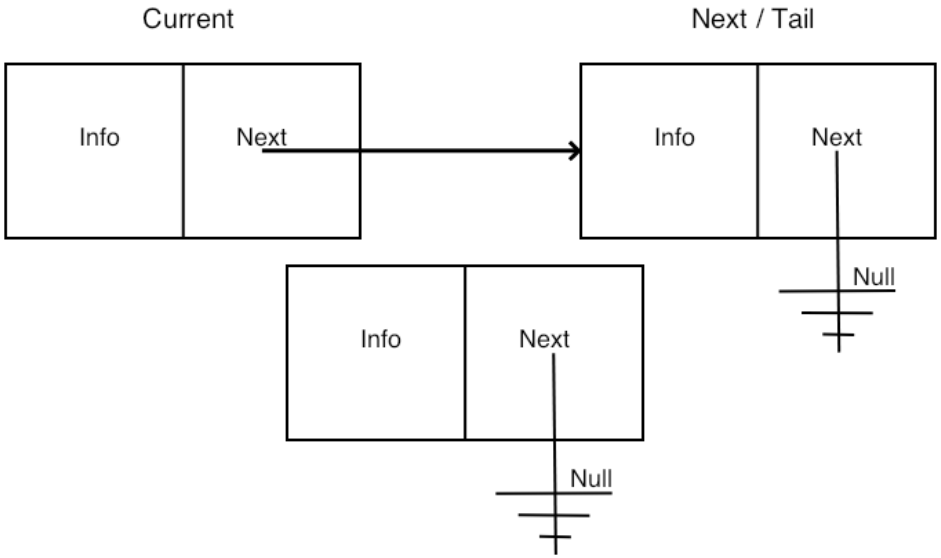
```
auto
  first = make_shared<Node>(23),
  second = make_shared<Node>(45),
  third = make_shared<Node>(32);
first->append(second);
first->append(third);
first->print();
```

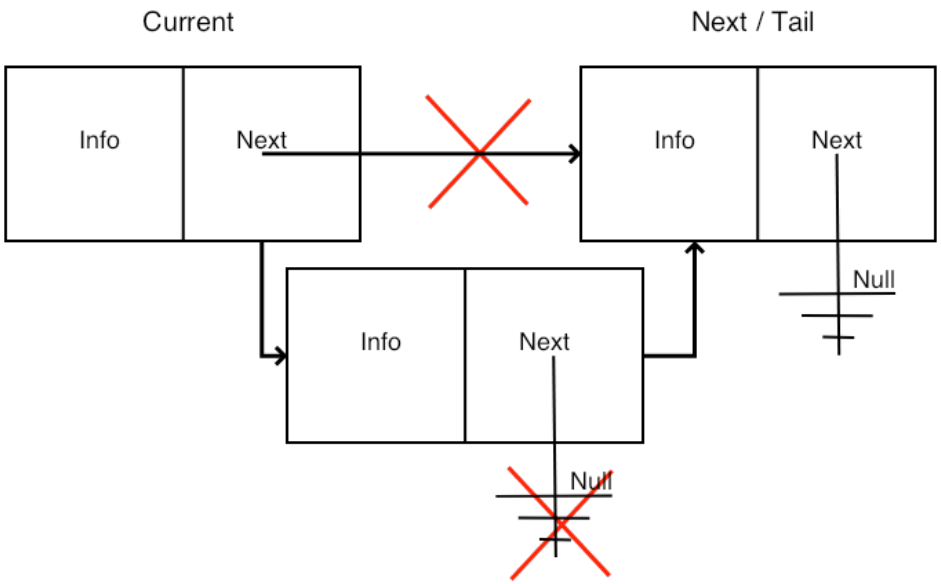
Output

```
[tree] append:

Append 23 & 45 gives
    <<23,45>>
Append 32 gives
    <<23,45,32>>
```

Insertion


Current
Next / Tail
Info
Next
Info
Next
Null
Info
Next
Null


Current
Next / Tail
Info
Next
Info
Next
Null
Info
Next
Null

Exercise 67

Write a recursive *insert* method that inserts a node in a list, such that the list stays sorted:

Code:

```
auto
  first = make_shared<Node>(23),
  second = make_shared<Node>(45),
  third = make_shared<Node>(32);
first->insert(second);
first->insert(third);
first->print();
```

Output

```
[tree] insert:

Insert 45 on 23 gives
     <<23,45>>
Insert 32 gives
    <<23,32,45>>
```

Assume that the new node always comes somewhere after the head node.

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.
if you write C, you'd better understand it.

172. 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. C style:

Code:

```
int i;
printf("address of i: %ld\n",
       (long)(&i));
printf(" same in hex: %lx\n",
       (long)(&i));
```

Output

```
[pointer] printfpoint:

address of i:
    140732781906948
 same in hex: 7
    ffee7794c04
```

and C++:

Code:

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

Output

```
[pointer] coutpoint:

address of i, decimal
    : 140732704635892
address of i, hex
    : 0x7ffee2de3bf4
```

173. 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;
```

174. Dereferencing

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

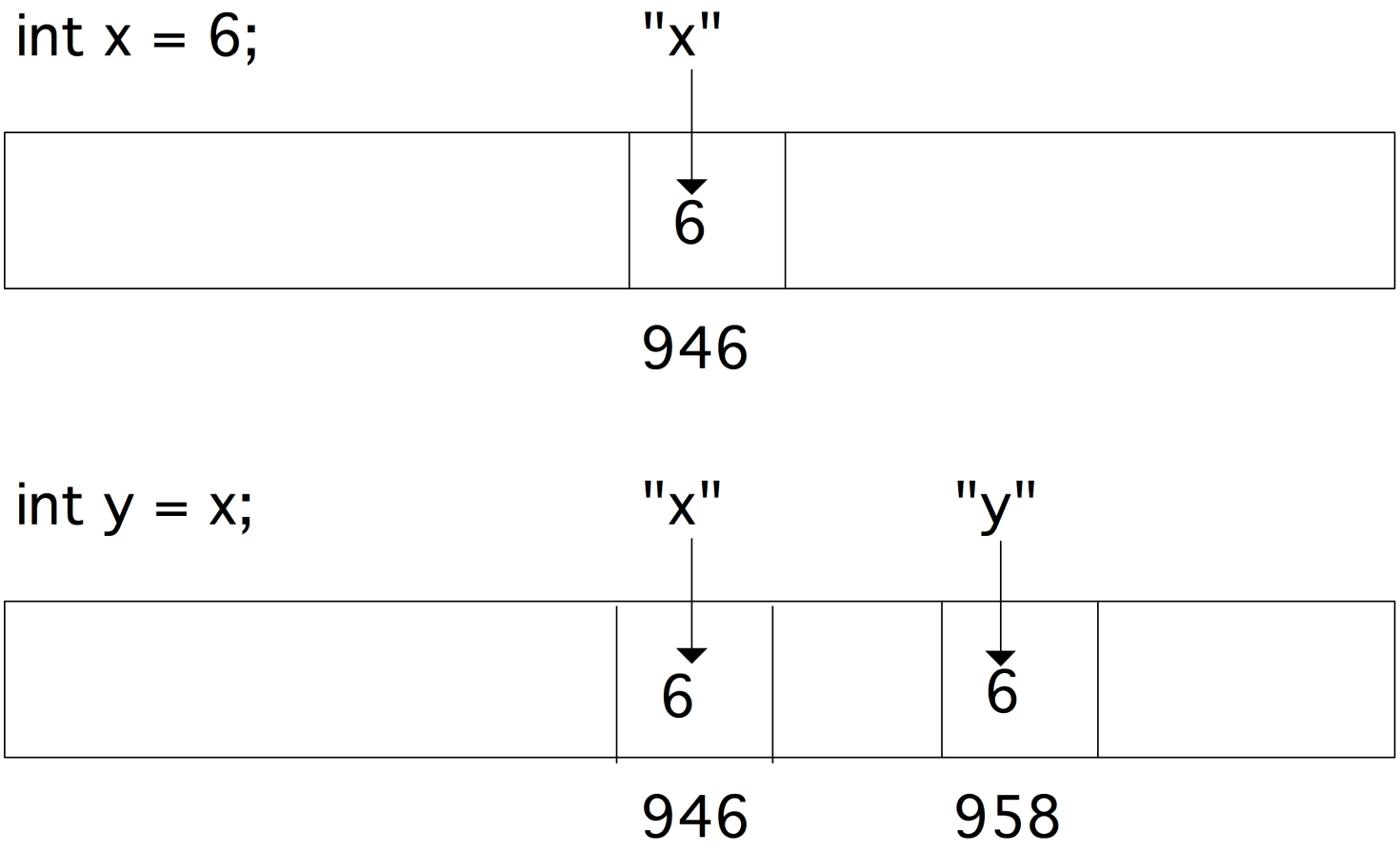
Code:

```
int i;
int* addr = &i;
i = 5;
cout << *addr << endl;
i = 6;
cout << *addr << endl;
```

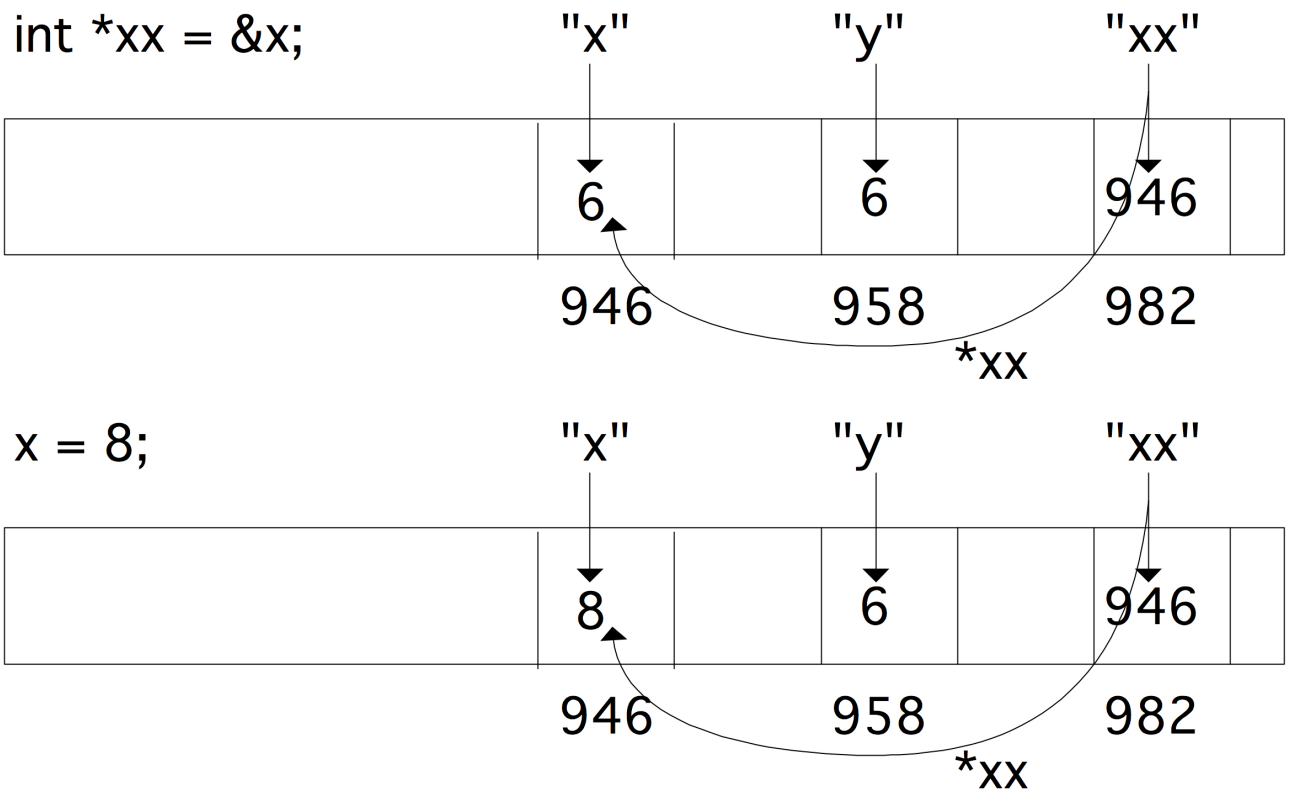
Output
[pointer] cintpointer:

```
5
6
```

175. illustration



176. illustration



177. Star stuff

Equivalent:

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

Addresses and parameter passing

178. 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;
}
```

179. 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 68

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

180. Array and pointer equivalence

Array and memory locations are largely the same:

Code:

```
double array[5] = {11,22,33,44,55};
double *addr_of_second = &(array[1]);
cout << *addr_of_second << endl;
array[1] = 7.77;
cout << *addr_of_second << endl;
```

Output

```
[pointer] arrayaddr:

22
7.77
```

181. Array and pointer equivalence

Array and memory locations are largely the same:

Code:

```
double array[5] = {11,22,33,44,55};
double *addr_of_second = &(array[1]);
cout << *addr_of_second << endl;
array[1] = 7.77;
cout << *addr_of_second << endl;
```

Output

```
[pointer] arrayaddr:

22
7.77
```

Multi-dimensional arrays

182. 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

183. Problem with static arrays

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

184. Declaration and allocation

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

(Size in doubles, not in bytes as in C)

185. De-allocation

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

```
delete(array);
```

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

186. 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.

187. Memory leaks

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

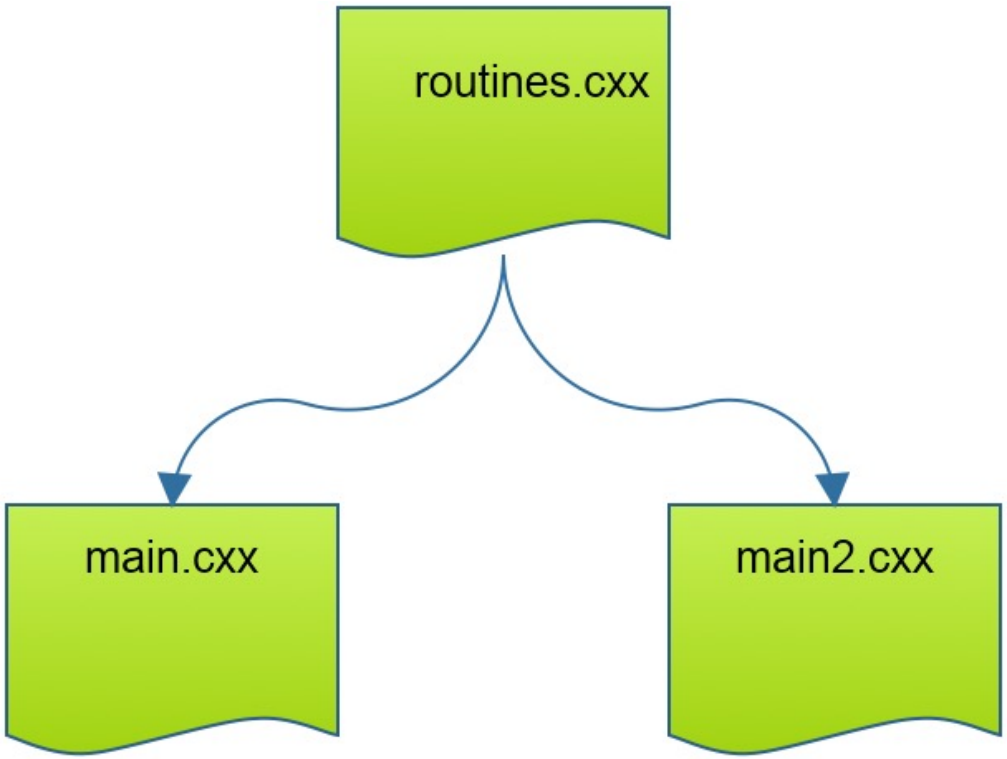
Every iteration reserves memory, which is never released: another memory leak.

Your code will run out of memory!

Prototypes, separate compilation

188. Include files

Reuse code by include it in multiple mains.



We will develop a better scenario.

189. Prototypes and forward declarations, 1

A first use of prototypes is forward declaration.

Some people like defining functions after the main:

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

versus before:

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

190. Prototypes and forward declarations, 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); }
```

191. Prototypes for separate compilation

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

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

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

192. 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
   ```

193. 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
```

194. Prototypes and header files

Header file contains only prototype:

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

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

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

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

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

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

195. Class prototypes

Header file:

```
class something {
private:
  int i;
public:
  double dosomething( int i, char c );
};
```

Implementation file:

```
double something::dosomething( int i, char c ) {
  // do something with i,c
};
```

Review quiz 15

For each of the following answer: is this a valid function definition or function prototype.

- `int foo();`
- `int foo() {};`
- `int foo(int) {};`
- `int foo(int bar) {};`
- `int foo(int) { return 0; };`
- `int foo(int bar) { return 0; };`

196. Make

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

(Also `Cmake`.)

Advanced topics

Lambda functions

197. A simple example

You can define a function and apply it:

```
double f(x) { return 2*x; }

y = f(3.7);
```

or you can appy the function recipe directly:

```
y = [] (double x) -> double { return 2*x; } (3.7);
```

198. Lambda syntax

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

- The square brackets are how you recognize a lambda
- Inputs: like function parameters
- Result type: can be omitted if compiler can deduce it;
- Definition: function body.

199. Assign lambda to variable

```
auto f = [] (double x) -> double { return 2*x; };
y = f(3.7);
z = f(4.9);
```

- This is a variable declaration.
- Uses `auto` for technical reasons
- See different approach below.

Exercise 69

The Newton method (see HPC book) for finding the zero of a function f, that is, finding the x for which $f(x) = 0$, can be programmed by supplying the function and its derivative:

```
double f(double x) { return x*x-2; };
double g(double x) { return 2*x; };
```

and the algorithm:

```
double x{1.};
while ( true ) {
  auto fx = f(x);
  cout << "f( " << x << " ) = " << fx << "\n";
  if (abs(fx)<1.e-10 ) break;
  x = x - fx/g(x);
}
```

Rewrite this code to use lambda functions for f and g.

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

200. Lambdas as parameter: the problem

Lambdas have a generated type, so you can not write a function that takes a lambda as argument.

```
void do_something( /* what? */ f ) {
    f(5);
}
int main() {
    do_something
      ( [] (double x) { cout << x; } );
}
```

201. Lambdas as parameter: the solution

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

With this, you can declare parameters by their signature:

```
double find_zero
   ( function< double(double)> f,
     function< double(double) > g ) {
```

Exercise 70

Rewrite the Newton exercise above to use a function with prototype

```
double root = find_zero( f,g );
```

Call the function directly with the lambda functions as arguments, that is, without assigning them to variables.

202. Capture parameter

Capture value and reduce number of arguments:

```
int exponent=5;
auto powerfive =
  [exponent] (float x) -> float {
    return pow(x,exponent); };
```

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

Code:

```
cout << "To the power "
     << exponent << endl;
for (float x=1.; x<=5.; x+=1.)
  cout << x << ":" << powerfive(x) <<
     endl;
```

Output

```
[func] lambdait:

To the power 5
1:1
2:32
3:243
4:1024
5:3125
```

Exercise 71

Extend the newton exercise to compute roots in a loop:

```
for (int n=2; n<=8; n++) {
  cout << "sqrt(" << n << ") = "
       << find_zero(
/* ... */
                    )
       << "\n";
```

Without lambdas, you would define a function

```
double to_the_nth( double x,int n );
```

However, the *find_zero* function takes a function of only a real argument. Use a capture to make *f* dependent on the integer parameter.

Exercise 72

You don't need the gradient as an explicit function: you can approximate it as

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

for some value of h.

Write a version of the root finding function

```
double find_zero( function< double(double)> f, double h=.001 );
```

that uses this. Do not reimplement the whole newton method: instead create a lambda for the gradient and pass it to the function `find_zero` you coded earlier.

203. Capture by reference

Capture variables are normally by value, use ampersand for reference. This is often used in *algorithm* header.

Code:

```
vector<int> moreints{8,9,10,11,12};
int count{0};
for_each
  ( moreints.begin(),moreints.end(),
    [&count] (int x) {
      if (x%2==0)
        count++;
    } );
cout << "number of even: " << count
   << endl;
```

Output

```
[stl] counteach:

number of even: 3
```

Namespaces

204. You have already seen namespaces

Safest:

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

Drastic:

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

Prudent:

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

205. Why not 'using namespace std' ?

This compiles, but should not:

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

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

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

This gives an error:

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

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

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

206. Defining a namespace

You can make your own namespace by writing

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

207. Namespace usage

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

or

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

or

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

or

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

Templates

208. Templated type name

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

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

209. Example: function

Definition:

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

Usage:

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

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

Exercise 73

Machine precision, or 'machine epsilon', is sometimes defined as the smallest number ϵ so that $1 + \epsilon > 1$ in computer arithmetic.

Write a templated function `epsilon` so that the following code prints out the values of the machine precision for the `float` and `double` type respectively:

Code:

```
float float_eps;
epsilon(float_eps);
cout << "Epsilon float: "
     << setw(10) << setprecision(4)
     << float_eps << endl;

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

Output

```
[template] eps:

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

210. Templated vector

The Standard Template Library (STL) contains in effect

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

Exceptions

211. Exception throwing

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

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

212. Catching an exception

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

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

213. 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!

214. 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
}
```

215. Catch any exception

Catch exceptions without specifying the type:

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

216. 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`.

217. Destructors and exceptions

The destructor is called when you throw an exception:

Code:

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

Output

```
[object] exceptdestruct:
calling the
    constructor
Inside the nested
    scope
calling the
    destructor
Exception caught
```

Auto

218. 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();
```

219. Type deduction in functions

Return type can be deduced in C++17:

```
auto equal(int i,int j) {
  return i==j;
};
```

220. Auto and references, 1

auto discards references and such:

Code:

```
A my_a(5.7);
auto get_data = my_a.access();
get_data += 1;
my_a.print();
```

Output

```
[auto] plainget:

data: 5.7
```

221. Auto and references, 2

Combine auto and references:

Code:

```
A my_a(5.7);
auto &get_data = my_a.access();
get_data += 1;
my_a.print();
```

Output

```
[auto] refget:

data: 6.7
```

222. Auto and references, 3

For good measure:

```
A my_a(5.7);
const auto &get_data = my_a.access();
get_data += 1; // WRONG does not compile
my_a.print();
```

223. Auto iterators

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

is actually short for:

```
for ( std::vector<int>
     iterator it=myvector.begin
    () ;
     it!=myvector.end() ; ++it
    )
   s += *it ; // note the deref
```

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

Random

224. Random floats

```
// seed the generator
std::random_device r;
// std::seed_seq ssq{r()};
// and then passing it to the engine does the same

// 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.);

cout << "first rand: " << distribution(generator) << endl;
```

225. 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
```

226. Poisson distribution

Another distribution is the Poisson distribution:

```
std::default_random_engine generator;
float mean = 3.5;
std::poisson_distribution<int> distribution(mean);
int number = distribution(generator);
```

227. Global engine

Wrong approach:

Code:

```
int nonrandom_int(int max) {
  std::default_random_engine engine;
  std::uniform_int_distribution<> ints
     (1,max);
  return ints(engine);
};
```

Output

```
[rand] nonrandom:

Three ints: 15, 15,
    15.
```

Good approach:

Code:

```
int realrandom_int(int max) {
  static  std::default_random_engine
     static_engine;
  std::uniform_int_distribution<> ints
     (1,max);
  return ints(static_engine);
};
```

Output

```
[rand] truerandom:

Three ints: 15, 98,
    70.
```

Libraries

228. Example: `cxxopts`

`https://github.com/jarro2783/cxxopts`

Find the `2.2.1` release.
Use `wget` or `curl` to download straight to the class machine.
Unpack it.

229. Cmake based installation

The `cxxopts-2.2.1` directory has a file `CMakeLists.txt`

```
mkdir build
cd build
cmake -D CMAKE_INSTALL_PREFIX:PATH=${HOME}/mylibs \
    ..
make
make install
```

(This is an 'in-source' build. I don't like it: prefer to have the build directory elsewhere to keep the source untouched.)

230. Let's use this library

```
#include "cxxopts.hpp"
int main() {
  cxxopts::Options
    options("programname", "Program description");
}
```

compile

```
icpc -o program source.cpp \
   -I/path/to//cxxopts/installdir/include
```

Can you compile and run this?

231. Commandline options

```
options.add_options()
  ("h,help","usage information")
  ("n,nsize","size of the thing",
   cxxopts::value<int>()->default_value("4096"))
   // et cetera
  ;
auto result = options.parse(argc, argv);
if (result.count("help")>0) {
  std::cout << options.help() << std::endl;
  return 0;
}
int array_size = result["nsize"].as<int>();
```

Write code to test this.

Can you add more types of options?

Software development

232. Programming and correctness

Find your favorite example of costly programming mistakes . . .

What to do about it?

- Never make mistakes.
- Prove that your program is correct.
- Test your program before deploying it.
- Handle errors as they occur.

Error handling

233. Assertions to catch logic errors

Sanity check on things 'that you just know are true':

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

Example:

```
x = sin(2.81);
y = x*x;
z = y * (1-y);
assert( z>=0. and z<=1. );
```

234. Using assertions

Check on valid input parameters:

```
#include <cassert>

// this function requires x<y
// it computes something positive
float f(x,y) {
  assert( x<y );
  return /* some result */;
}
```

Check on valid results:

```
float positive_outcome = f(x,y);
assert( positive_outcome>0 );
```

235. Example

```
int collatz_next( int current ) {
  assert( current>0 );
  int next{-1};
  if (current%2==0) {
    next = current/2;
    assert(next<current);
  } else {
    next = 3*current+1;
    assert(next>current);
  }
  return next;
}
```

236. Use assertions during development

Assertions are disabled by

```
#define NDEBUG
```

before the include.

You can pass this as compiler flag:
`icpc -DNDEBUG yourprog.cxx`

237. Exceptions

Not every error is fatal:

$$\text{Exception} \equiv \begin{cases} \text{'this should not happen'} \\ \text{but we can handle it} \end{cases}$$

1. recover from the problem
2. graceful exit

238. Exceptions

Have you seen the following?

Code:

```
vector<float> x(5);
x.at(5) = 3.14;
```

Output

```
[except] boundthrow:

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

The STL can generate many exceptions.

- You can let your program crash, and start debugging
- You can try to catch and handle them yourself.

239. Exception structure

Code with problem:

```
if ( /* some problem */ )
  throw(5);
  /* or: throw("error"); */
```

```
try {
  /* code that can go wrong */
} catch (...) { // literally
    three dots!
  /* code to deal with the
    problem */
}
```

240. Exceptions

Assume a routine only works for certain values, and you want to generate an error if called with an inappropriate value.

```
double compute_root(double x) {
  if (x<0) throw(1);
  return sqrt(x);
}
int main() {
  try {
    y = compute_root(x);
  } catch (...) {
    /* handle error */
    cout << "Root failed, using default\n";
    y = 0;
  }
```

See book for more details.

Unit testing and test-driven development (TDD)

241. 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 . . .

242. 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.

243. 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

244. 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

245. 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.

246. Unit testing frameworks

Testing is important, so there is much software to assist you.

Popular choice with C++ programmers: Catch2
`https://github.com/catchorg`

247. Toy example

Function and tester:

```
double f(int n) { return n*n+1; }

#define CATCH_CONFIG_MAIN
#include "catch2/catch_all.hpp"

TEST_CASE( "test that f always returns positive" ) {
  for (int n=0; n<1000; n++)
    REQUIRE( f(n)>0 );
}
```

(accept the `define` and `include` as magic)

248. Compiling toy example

```
icpc -o tdd tdd.cxx \
    -I${TACC_CATCH2_INC} -L${TACC_CATCH2_LIB} \
    -lCatch2Main -lCatch2
```

- Files:

  ```
  icpc -o tdd tdd.cxx
  ```

- Path to include and library files:

  ```
  -I${TACC_CATCH2_INC} -L${TACC_CATCH2_LIB}
  ```

- Libraries:

  ```
  -lCatch2Main -lCatch2
  ```

Exercise 74: Extend the toy example

1. Write a function

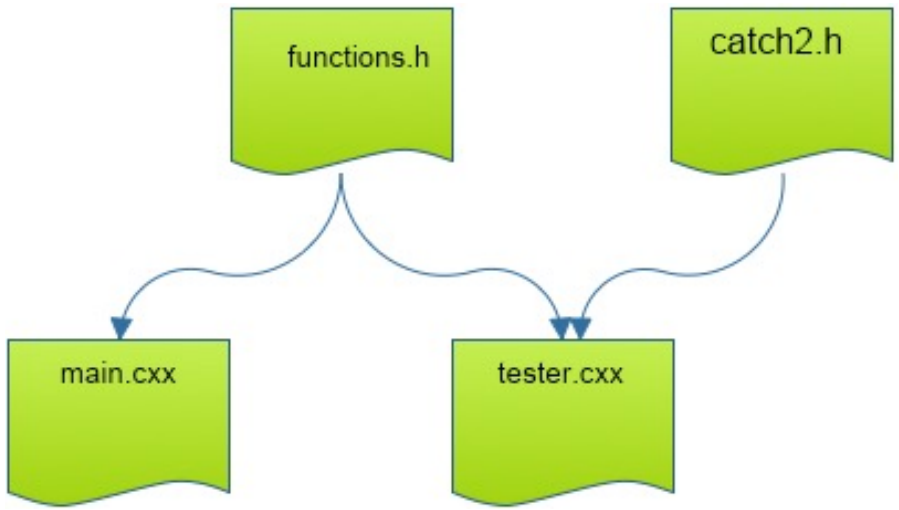
   ```
   double f(int n) { /* .... */ }
   ```

   with values in the range $(0, 1)$.
2. Write a unit test for this.

You can base this off the file `tdd.cxx` in the repository

249. Realistic setup

- All program functionality in a 'library' file
- Main program really short
- Tester file with only tests.
- (Tester also needs the `catch2` stuff included)



250. Slightly realistic example

Example: we use a function that

- only works for positive inputs;
- returns input $+1$.

Program that uses this:

```
#include "functions.h"
int main() {
  for ( int i=10; i>-1; i-- )
    cout << "One more than the positive number "
         << i << " is "
         << increment_positive_only(i)
         << "\n";
```

Note the include file!

251. Function to be developed

We know the structure:

```
int increment_positive_only( int i ) {
  // this function returns one more than the inpuut
  // input has to be positive, error otherwise
  /* ... */
}
```

function body to be developed.

252. Functionality testing

File `tester.cxx`:

Same include file for the functionality;
the testing framework creates its own main.

```
#include "functions.h"

#define CATCH_CONFIG_MAIN
#include "catch2/catch_all.hpp"

TEST_CASE( "test the increment function" ) {
  /* ... */
}
```

253. Compiling the tester

One-line solution:

```
icpc -o tester test_main.cxx \
    -I${TACC_CATCH2_INC} -L${TACC_CATCH2_LIB} \
    -lCatch2Main -lCatch2
```

254. Compile and link options

Variables for a Makefile:

```
INCLUDES = -I${TACC_CATCH2_INC}
EXTRALIBS = -L${TACC_CATCH2_LIB} -lCatch2Main -lCatch2
```

Exercise 75: File structure

Make three files:

1. Include file with the functions.
2. Main program that uses the functions.
3. Tester main file, contents to be determined.

255. Correctness through 'require' clause

Tests go in `tester.cxx`:

```
TEST_CASE( "test that f always returns positive" ) {
  for (int n=0; n<1000; n++)
    REQUIRE( f(n)>0 );
}
```

- *TEST_CASE* acts like independent program.
- REQUIRE is like assert but more sophisticated
- Can contain (multiple) tests for correctness.

256. Output for failing tests

Run the tester:

```
-------------------------------------------------------------------
test the increment function
-------------------------------------------------------------------
test.cxx:25
...............................................................

test.cxx:29: FAILED:
  REQUIRE( increment_positive_only(i)==i+1 )
with expansion:
  1 == 2

===================================================================
test cases: 1 | 1 failed
assertions: 1 | 1 failed
```

257. 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( "function fails for " << n );
    REQUIRE( f(n)>0 );
}
```

Exercise 76: Positive tests

Continue with the example of slide 418:
add a positive `TEST_CASE`

```
for (int i=1; i<10; i++)
  REQUIRE( increment_positive_only(i)==i+1 );
```

Make the function satisfy this test.

258. Test for exceptions

Supppose function `g(n)`

- succeeds for input $n > 0$
- fails for input $n \leq 0$:
  throws exception

```
TEST_CASE( "test that g only works for positive" ) {
  for (int n=-100; n<+100; n++)
    if (n<=0)
      REQUIRE_THROWS( g(n) );
    else
      REQUIRE_NOTHROW( g(n) );
}
```

Exercise 77: Negative tests

Make sure your function throws an exception at illegal inputs:

```
for (int i=0; i>-10; i--)
  REQUIRE_THROWS( increment_positive_only(i) );
```

259. Tests with code in common

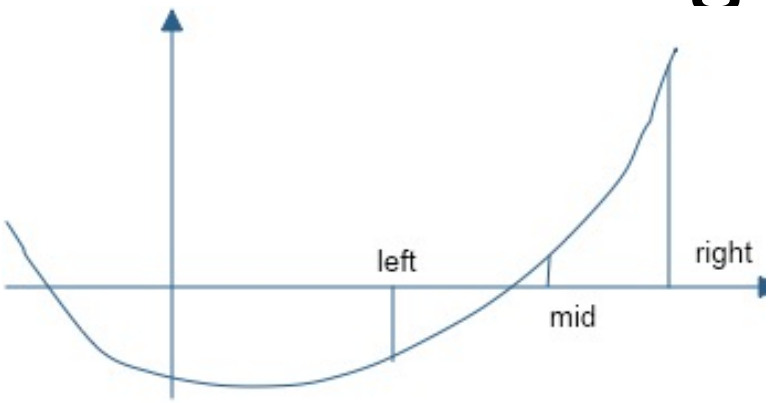
Use SECTION if tests have intro/outtro in common:

```
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)

TDD example: Bisection

260. 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.

261. Coefficient handling

$$f(x) = c_d x^d + \cdots + c_1 x^1 + c_0$$

We implement this by storing the coefficients in a `vector<double>`.
Proper:

```
TEST_CASE( "coefficients are polynomial","[1]" ) {
  auto coefficients = set_coefficients();
  REQUIRE( coefficients.size()>0 );
  REQUIRE( coefficients.front()!=0. );
}
```

Exercise 78: Proper polynomials

Write a routine *set_coefficients* that constructs a vector of coefficients:

```
vector<double> coefficients = set_coefficients();
```

and make it satisfy the above conditions.

At first write a hard-coded set of coefficients, then try reading them from the command line.

Exercise 79: One test for properness

Write a test *proper_polynomial* and write unit tests for it, both passing and failing.

262. Test on polynomials evaluation

```
// correct interpretation: 2x^2 + 1
vector<double> second{2,0,1};
REQUIRE( proper_polynomial(second) );
REQUIRE( evaluate_at(second,2) == Catch::Approx(9) );
// wrong interpretation: 1x^2 + 2
REQUIRE( evaluate_at(second,2) != Catch::Approx(6) );
```

Exercise 80: Implementation

Write a function *evaluate_at* which computes

$$y \leftarrow f(x).$$

For bonus points, look up Horner's rule and implement it.

Exercise 81: Odd degree polynomials only

With odd degree you can always find bounds x_-, x_+.
Reject even degree polynomials:

```
if ( not is_odd(coefficients) ) {
  cout << "This program only works for odd-degree polynomials\n";
  exit(1);
}
```

Gain confidence by unit testing:

```
vector<double> second{2,0,1}; // 2x^2 + 1
REQUIRE( not is_odd(second) );
vector<double> third{3,2,0,1}; // 3x^3 + 2x^2 + 1
REQUIRE( is_odd(third) );
```

Exercise 82: Find bounds

Write a function *find_outer* 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?)
Unit test:

```
right = left+1;
vector<double> second{2,0,1}; // 2x^2 + 1
REQUIRE_THROWS( find_outer(second,left,right) );
vector<double> third{3,2,0,1}; // 3x^3 + 2x^2 + 1
REQUIRE_NOTHROW( find_outer(third,left,right) );
REQUIRE( left<right );
```

How would you test the function values?

Exercise 83: Put it all together

Make this call work:

```
auto zero = find_zero(coefficients,left,right);
cout << "Found root " << zero
     << " with value " << evaluate_at(coefficients,zero) << "\n";
```

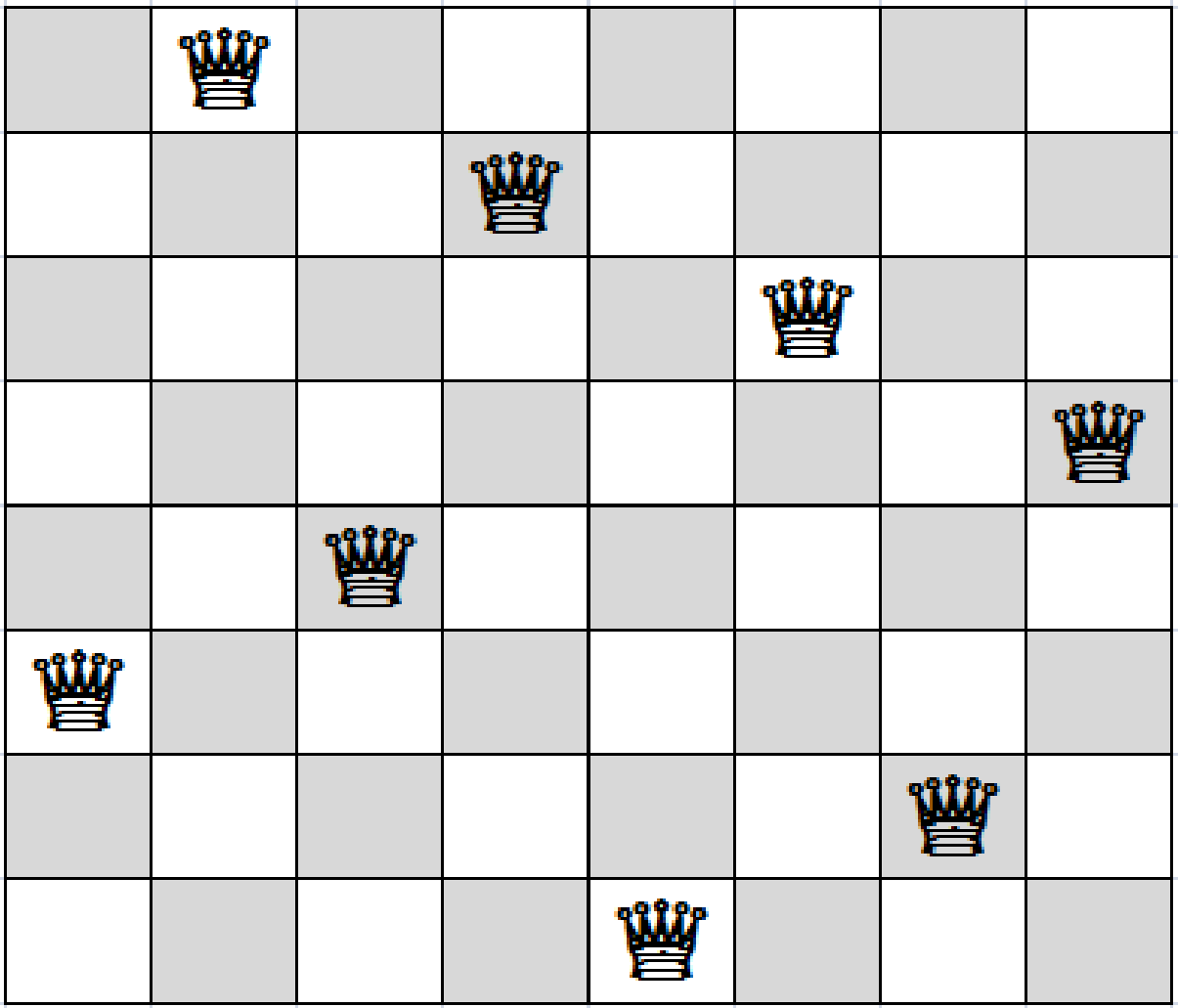
Add an optional precision argument to the root finding function.

Design unit tests, including on the precision attained, and make sure your code passes them.

Eight queens problem

263. Problem statement

Can you place eight queens on a chess board so that no pair threatens each other?

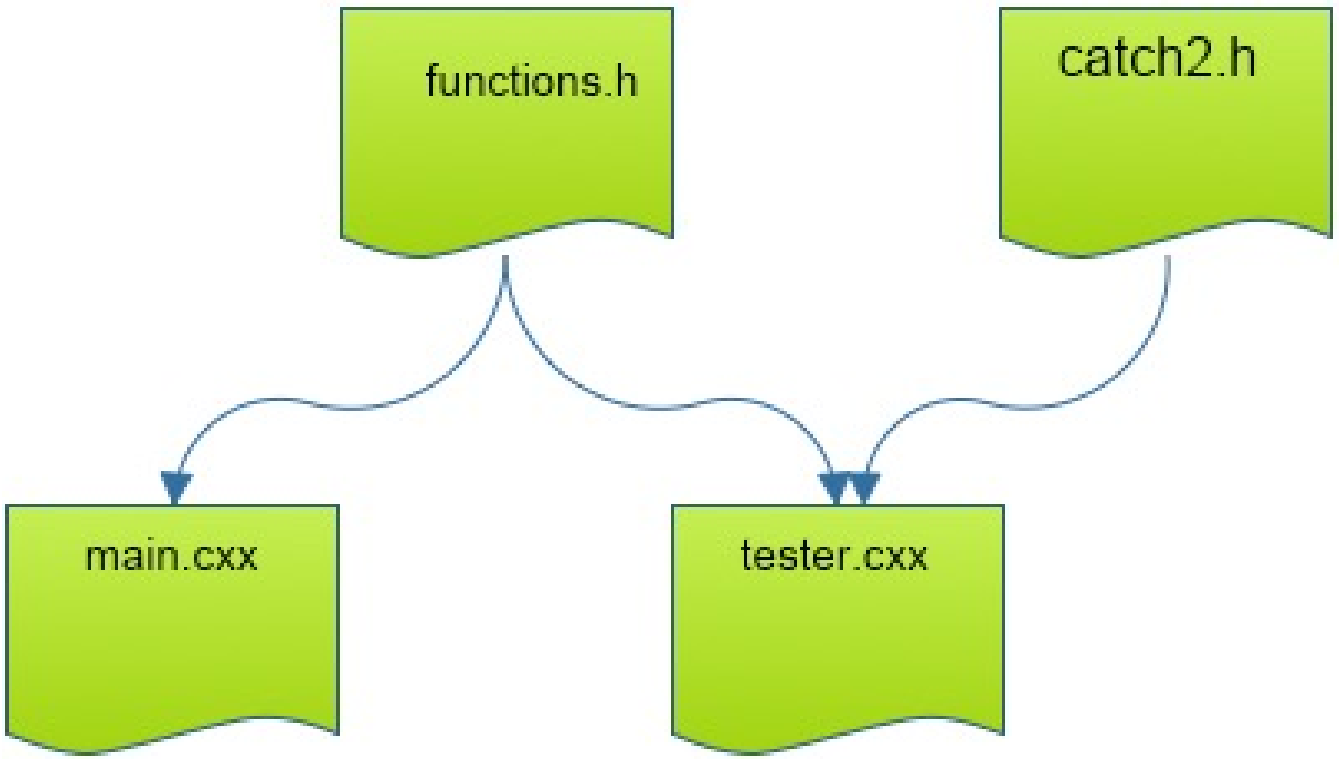


264. Sort of test-driven development

You will solve the 'eight queens' problem by

- designing tests for the functionality
- then implementing it

265. File structure



266. Basic object design

Object constructor of an empty board:

```
board(int n);
```

Test how far we are:

```
int next_row_to_be_filled() const;
```

First test:

```
TEST_CASE( "empty board" ) {
  constexpr int n=10;
  board empty(n);
  REQUIRE( empty.next_row_to_be_filled()==0 );
}
```

Exercise 84: Board object

Start writing the *board* class, and make it pass the above test.

Exercise 85: Board method

Write a method for placing a queen on the next row,

```
void place_next_queen_at_column(int i);
```

and make it pass this test (put this in a *TEST_CASE*):

```
auto one(empty);
REQUIRE_THROWS( one.place_next_queen_at_column(-1) );
REQUIRE_THROWS( one.place_next_queen_at_column(n) );
REQUIRE_NOTHROW( one.place_next_queen_at_column(0) );
REQUIRE( one.next_row_to_be_filled()==1 );
```

Exercise 86: Test for collisions

Write a method that tests if a board is collision-free:

```
bool feasible() const;
```

This test has to work for simple cases to begin with. You can add these lines to the above tests:

```
REQUIRE( empty.feasible() );

REQUIRE( one.feasible() );

auto collide(one);
collide.place_next_queen_at_column(0);
REQUIRE( not collide.feasible() );
```

Exercise 87: Test full solutions

Make a second constructor to 'create' solutions:

```
board( vector<int> cols );
```

Now we test small solutions:

```
board five( {0,3,1,4,2} );
REQUIRE( five.feasible() );
```

Exercise 88: No more delay: the hard stuff!

Write a function that takes a partial board, and places the next queen:

```
optional<board> place_queen(const board& current);
```

Test that the last step works:

```
board almost( {1,3,0,board::magic::empty} );
auto solution = place_queen(almost);
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 89: Test that you can find solutions

Test that there are no 3×3 solutions:

```
TEST_CASE( "no 3x3 solutions" ) {
  board three(3);
  auto solution = place_queen(three);
  REQUIRE( not solution.has_value() );
}
```

but 4×4 solutions do exist:

```
TEST_CASE( "there are 4x4 solutions" ) {
  board four(4);
  auto solution = place_queen(four);
  REQUIRE( solution.has_value() );
}
```

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.

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:

$$\begin{pmatrix} y_1 \\ \dots \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} A_1 \\ \dots \\ A_2 \\ A_3 \end{pmatrix} (x)$$

so

$$y_1 = A_1 x$$

is true

Then the update step, and

After

$$\begin{pmatrix} y_1 \\ y_2 \\ \dots \\ y_3 \end{pmatrix} = \begin{pmatrix} A_1 \\ A_2 \\ \dots \\ A_3 \end{pmatrix} (x)$$

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

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:

$$(y) = \left(\begin{array}{ccc} A_1 & \vdots & A_2 \quad A_3 \end{array}\right) \begin{pmatrix} x_1 \\ \dots \\ x_2 \\ x_3 \end{pmatrix}$$

Then the update step, and
After

$$(y) = \left(\begin{array}{ccc} A_1 \quad A_2 & \vdots & A_3 \end{array}\right) \begin{pmatrix} x_1 \\ x_2 \\ \dots \\ x_3 \end{pmatrix}$$

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 $c = 1, n$
    $y \leftarrow y + A_{*,c}x_c$