C++ Intro Catchup

Victor Eijkhout, Susan Lindsey

Fall 2022

last formatted: September 1, 2022

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;

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

and compile it. Intel compiler:

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

Run this program (it gives no output):

```
./zeroprogram
```

2. Anatomy of the compile line

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

Input/Output and strings

3. Terminal output

Terminal (console) output with `cout`:

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

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

4. Terminal input

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

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

5. 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.

6. 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";
```

7. Concatenation

Strings can be *concatenated*:

Code:

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

Output

```
[string] stringadd:

foo bar: 7
```

Conditionals

8. 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.

9. Complicated conditionals

Chain:

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

Nest:

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

10. 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.

Exercise 2

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.

For loops

11. 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!

12. 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.

13. 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) { ... }
```

14. 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 3

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.

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

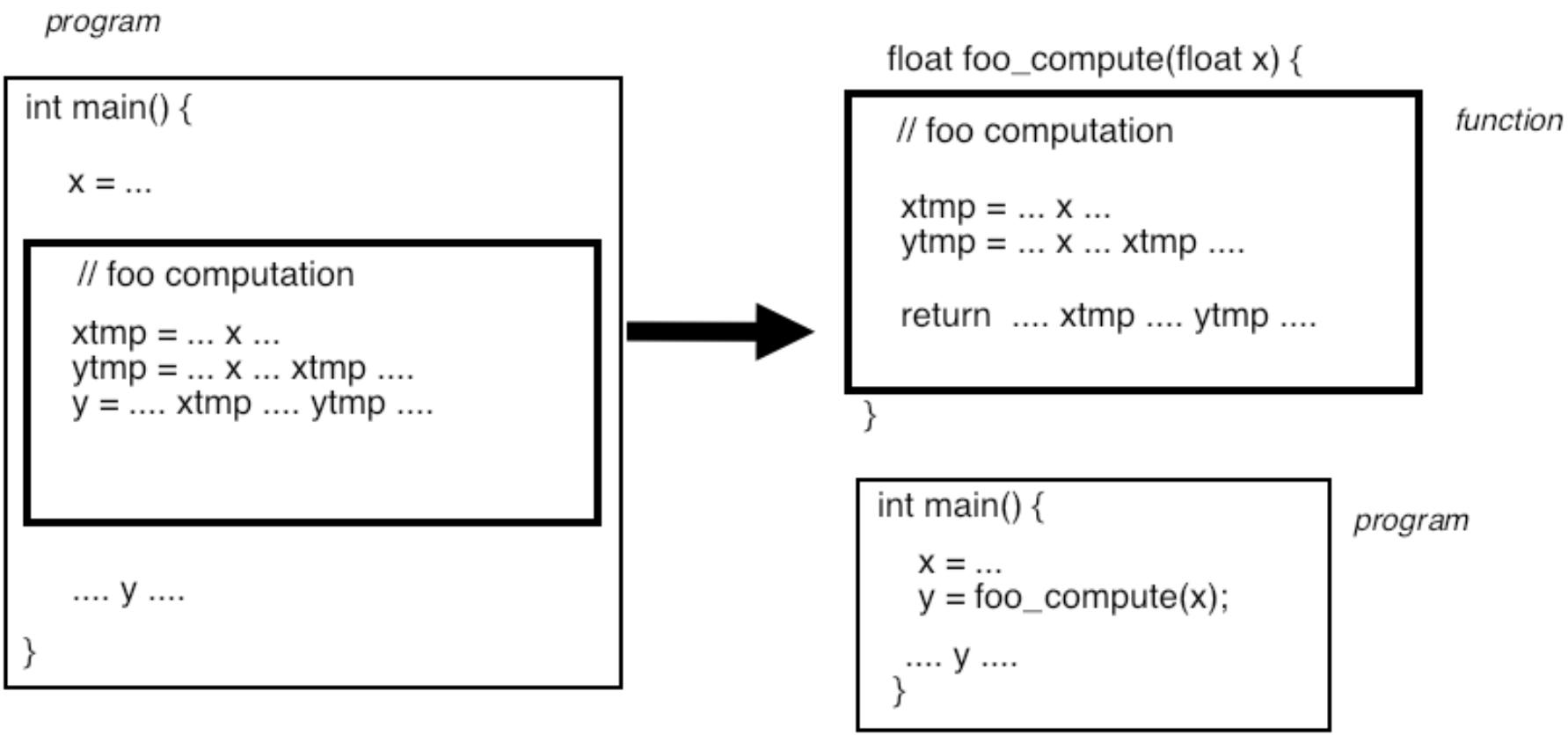
$$7 \to 22 \to 11 \to 34 \to 17 \to 52 \to 26 \to 13 \to 40 \to 20 \to 10 \to 5 \cdots$$

(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.

Function basics

15. Introducing a function



16. 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

Project Exercise 4

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?

Parameter passing

17. 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'

18. 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 << '\n';
  other = squared(number);
  cout << "Output var is: "
       << other << '\n';
  cout << "Input var is now: "
       << number << '\n';
```

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.

19. 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 << '\n';
i *= 2;
cout << i << "," << ri << '\n';
ri -= 3;
cout << i << "," << ri << '\n';
```

Output
[basic] ref:

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

(You will not use references often this way.)

20. 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
}
```

21. Pass by reference example 1

Code:

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

  int var = 0;
  f(var);
  cout << var << '\n';
```

Output

```
[basic] setbyref:

5
```

Compare the difference with leaving out the reference.

Exercise 5

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

Code:

```
cout << i << "," << j << '\n';
swap(i,j);
cout << i << "," << j << '\n';
```

Output

```
[func] swap:

1,2
2,1
```

More about functions

22. 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
```

Vectors

23. 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};
  auto halfloats = {0.5f, 1.5f, 2.5f};
  cout << evens.at(0) << '\n';
  return 0;
}
```

24. 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.

25. 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] << '\n';
```

```
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) << '\n';
```

```
Output
[array] assignatfun:

4,8
```

Safer, slower.

(Remember Knuth about optimization.)

26. Range over elements

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

```
for ( float e : my_data )
  // statement about element e
for ( auto e : my_data )
  // 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)" << '\n';
```

Output

```
[array] dynamicmax:

Max: 6 (should be 6)
```

27. 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) << '\n';
```

Output

```
[array] vectorrangeref:

6.6
```

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

28. Indexing the elements

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

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

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

Code:

```
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 <<
     '\n';
```

Output

```
[array] vecidxmax:

Max: 6.6 at index: 3
```

29. 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) << '\n';
```

Output

```
[array] vectorcopy:

3.5,7
```

30. Vector methods

A vector is an object, with methods.

Given `vector<sometype> x`:

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

Vectors and functions

31. 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.

32. 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) << '\n';
```

Output

```
[array] vectorpassref:

4.6
```

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

33. 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() <<
     '\n';
  cout << "zero element check: " <<
     x1.at(0) << '\n';
```

Output

```
[array] vectorreturn:

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

Exercise 6

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

input:

```
5,6,2,4,5
```

output:

```
5,5
6,2,4
```

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