Functions

Victor Eijkhout, Susan Lindsey

Fall 2025

last formatted: September 4, 2025

Reference material

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

Function basics

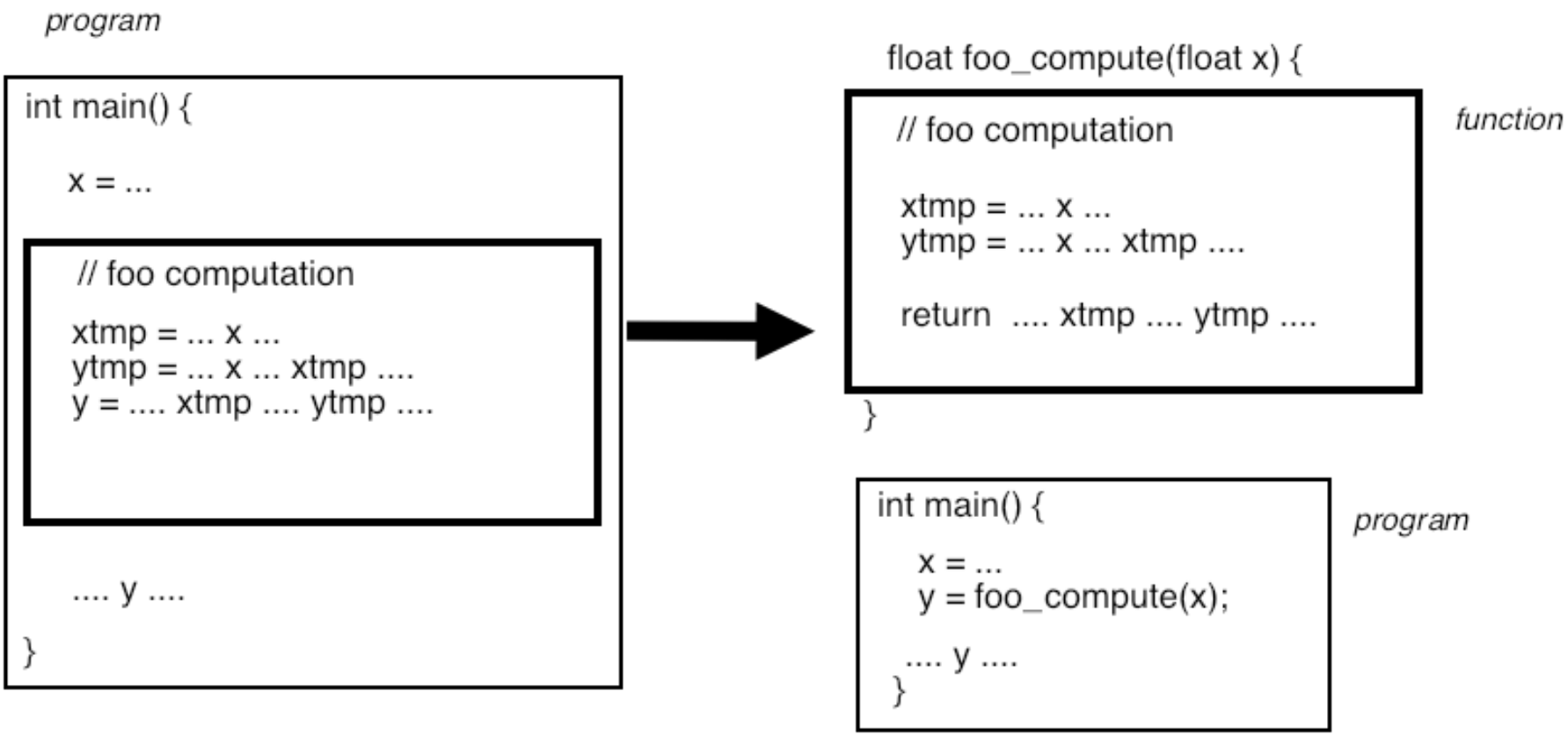
1. Why functions?

Functions are an abstraction mechanism.

- Isolate code fragment with clear function.
- Turn into *subprogram*: this is the function *definition*.
- Use by single line: this is the function *call*.
- Easy way to use code in multiple places.
- Introduce a **name** for a section of code;
  introduced a new level of programming vocabulary.

2. Introducing a function

Transforming a single file to main and function:



3. Code reuse

Suppose you do the same computation twice:

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

With a function this can be replaced by:

```
1 double computation(double in) {
2   return .... computation from `in' ....
3 }
4
5 y = computation(x);
6 w = computation(v);
```

4. Code reuse

Example: multiple norm calculations:

Repeated code:

```
1 float s = 0;
2 for (int i=0; i<x.size(); i++)
3   s += abs(x[i]);
4 cout << "One norm x: " << s << endl
      ;
5 s = 0;
6 for (int i=0; i<y.size(); i++)
7   s += abs(y[i]);
8 cout << "One norm y: " << s << endl
      ;
```

becomes:

```
1 float OneNorm( vector<float> a ) {
2   float sum = 0;
3   for (int i=0; i<a.size(); i++)
4     sum += abs(a[i]);
5   return sum;
6 }
7 int main() {
8   ... // stuff
9   cout << "One norm x: "
10        << OneNorm(x) << endl;
11  cout << "One norm y: "
12        << OneNorm(y) << endl;
```

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

5. Program without functions

Example: zero-finding through bisection.

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

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

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

Code:

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

Output:

```
1 Zero at: 1.4375
```

6. Introducing functions, step 1

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

```
1 // func/bisect2.cpp
2 float f(float x) {
3   return x*x*x - x*x-1;
4 };
```

Used in main:

```
1 // func/bisect2.cpp
2 while (right-left>.1) {
3   mid = (left+right)/2.;
4   float fmid = f(mid);
5   if (fmid<0)
6     left = mid;
7   else
8     right = mid;
9 }
```

7. Introducing functions, step 2

Function:

```
1 // func/bisect3.cpp
2 float f(float x) {
3   return x*x*x - x*x-1;
4 };
5 float find_zero_between
6     (float l,float r) {
7   float mid;
8   while (r-l>.1) {
9     mid = (l+r)/2.;
10    float fmid = f(mid);
11    if (fmid<0)
12      l = mid;
13    else
14      r = mid;
15  }
16  return mid;
17 };
```

New main:

```
1 // func/bisect3.cpp
2 int main() {
3   float left{0.},right{2.};
4   float zero =
5     find_zero_between(left,right);
6   cout << "Zero happens at: "
7        << zero << '\n';
8   return 0;
9 }
```

8. Single Responsibility Principle

- Make sure each function has one, clearly delineated purpose.
- Give it a good name that reflects that.
- Do not let a function do unrelated things.

Quiz 1

True or false?

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

9. Declaration first, definition last

Some people like the following style of defining a function:

```
1 // declaration before main
2 int my_computation(int);
3
4 int main() {
5   int result;
6   result = my_computation(5);
7   return 0;
8 };
9
10 // definition after main
11 int my_computation(int i) {
12    return i+3;
13 }
```

This is purely a matter of style.

10. Anatomy of a function definition

```
1 void write_to_file(int i,double x) { /* ... */ }
2 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.

11. Function call

The function call `y=f(x)` with

```
1 float f( float x ) { /* stuff */ return fx; }
```

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`.)

Quiz 2

True or false?

- A function can have only one input
- A function can have only one return result
- A void function can not have a `return` statement.

Programming Project Exercise 1

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

```
1 int main() {
2   bool isprime;
3   isprime = is_prime(13);
```

Write a main program that reads the number in, and prints the value of the boolean. (How is the boolean rendered? See section Textbook, section 12.4.3.)

Does your function have one or two `return` statements? Can you imagine what the other possibility looks like? Do you have an argument for or against it?

Programming Project Exercise 2

Take your prime number testing function *is_prime*, and use it to write a program that prints multiple primes:

- Read an integer *how_many* from the input, indicating how many (successive) prime numbers should be printed.
- Print that many successive primes, each on a separate line.
- (Hint: keep a variable *number_of_primes_found* that is increased whenever a new prime is found.)

12. Square roots by Newton's method

Suppose you have a positive value y and you want want to compute $x = \sqrt{y}$. This is equivalent to finding the zero of

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

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

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

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

Optional exercise 3

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

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

Parameter passing

Reference material

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

13. 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'

14. Pass by value example

Note that the function alters its parameter x:

Code:

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

Output:

```
1 /bin/sh: line 0:
      ↪eval: -g:
      ↪invalid option
2 eval: usage: eval
      ↪[arg ...]
3 make[1]: ***
      ↪[passvalue.o]
      ↪Error 2
```

but the argument in the main program is not affected.

15. Reference

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

Code:

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

Output:

```
1 5,5
2 10,10
3 7,7
```

(You will not use references often this way.)

16. Parameter passing by reference

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

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

17. Results other than through return

Also good design:

- Return no function result,
- or return exit 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’.

18. Pass by reference example 1

Code:

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

Output:

```
1 5
```

Compare the difference with leaving out the reference.

19. Pass by reference example 2

```
1 bool can_read_value( int &value ) {
2   // this uses functions defined elsewhere
3   int file_status = try_open_file();
4   if (file_status==0)
5     value = read_value_from_file();
6   return file_status==0;
7 }
8
9 int main() {
10  int n;
11  if (!can_read_value(n)) {
12    // if you can't read the value, set a default
13    n = 10;
14  }
15  ..... do something with 'n' ....
```

We will learn better ways!

Exercise 4

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

Code:

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

Output:

```
1 /bin/sh: line 0:
      ↪eval: -g:
      ↪invalid option
2 eval: usage: eval
      ↪[arg ...]
3 make[1]: ***
      ↪[swap.o] Error
      ↪2
```

Exercise 5

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

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

Code:

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

Output:

```
1 /bin/sh: line 0:
      ↪eval: -g:
      ↪invalid option
2 eval: usage: eval
      ↪[arg ...]
3 make[1]: ***
      ↪[divisible.o]
      ↪Error 2
```

Exercise 6

Write a function with inputs x, y, θ that alters x and y corresponding to rotating the point (x, y) over an angle θ.

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

Your code should behave like:

Code:

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

Output:

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

Recursion

Reference material

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

20. 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

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

```
1 int factorial( int n ) {
2   if (n==1)
3     return 1;
4   else
5     return n*factorial(n-1);
6 }
```

Exercise 7

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 8

It is possible to define multiplication as repeated addition:

Code:

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

Output:

```
1 /bin/sh: line 0:
      ↪eval: -g:
      ↪invalid option
2 eval: usage: eval
      ↪[arg ...]
3 make[1]: ***
      ↪[mult.o] Error
      ↪2
```

Extend this idea to define powers as repeated multiplication.

Exercise 9

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

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

Extend the code of exercise 8 to implement this.

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

Exercise 10

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

21. Default arguments

Functions can have default argument(s):

```
1 double distance( double x, double y=0. ) {
2   return sqrt( (x-y)*(x-y) );
3 }
4   ...
5   d = distance(x); // distance to origin
6   d = distance(x,y); // distance between two points
```

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

22. Useful idiom

Don't trace a function unless I say so:

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

23. Polymorphic functions

You can have multiple functions with the same name:

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

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

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

Scope

Reference material

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

24. Lexical scope

Visibility of variables

```
1 int main() {
2   int i;
3   if ( something ) {
4     int j;
5     // code with i and j
6   }
7   int k;
8   // code with i and k
9 }
```

25. Shadowing

```
1 int main() {
2   int i = 3;
3   if ( something ) {
4     int i = 5;
5   }
6   cout << i << endl; // gives 3
7   if ( something ) {
8     float i = 1.2;
9   }
10  cout << i << endl; // again 3
11 }
```

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

Exercise 11

What is the output of this code?

```
1 // basic/shadowfalse.cpp
2 bool something{false};
3 int i = 3;
4 if ( something ) {
5   int i = 5;
6   cout << "Local: " << i << '\n';
7 }
8 cout << "Global: " << i << '\n';
9 if ( something ) {
10  float i = 1.2;
11  cout << i << '\n';
12  cout << "Local again: " << i << '\n';
13 }
14 cout << "Global again: " << i << '\n';
```

26. Life time vs reachability

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

```
1 void f() {
2   ...
3 }
4 int main() {
5   int i;
6   f();
7   cout << i;
8 }
```