Functions

Victor Eijkhout, Susan Lindsey

COE 322 Fall 2021

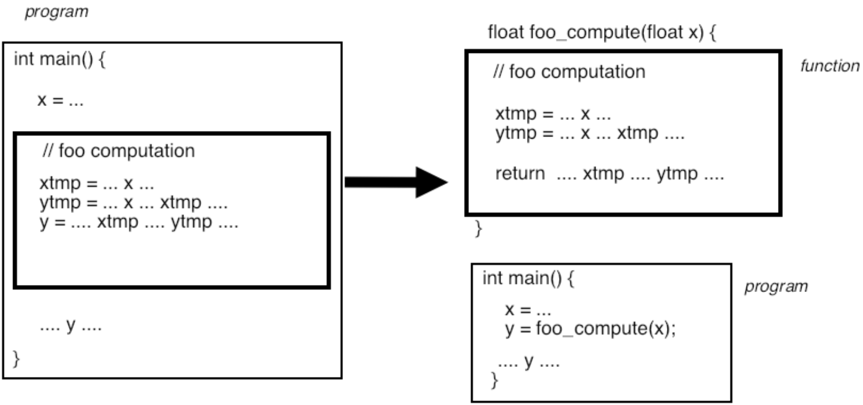
Function basics

1 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.

2 Introducing a function



3 Program without functions

Example: zero-finding through bisection.

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

Step 1: 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
```

4 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;
}
```

5 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 1

Take the source of the previous example, and introduce 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`

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

6 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

7 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);
```

8 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 1

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"`

Declaration vs definition

The compiler needs to know about a function before you can use it

Solution 1: define before use

```
double f(double x) {
  return x+1; }

int main() {
  double x=1;
  double y = f(x);
}
```

Solution 2: declare before use, define later

```
double f(double);

int main() {
  double x=1;
  double y = f(x);
}

double f(double x) {
  return x+1; }
```

9 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.

10 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 2

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 2

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 3

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 4

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'.)

11 Background Square roots through Newto

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

$$x_{\mathrm{next}} = x - f_y(x)/f_y'(x)$$

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

Optional exercise 5

- 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

12 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'

13 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.

14 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.)

15 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
}
```

16 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'.

17 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.

18 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` later; **??**.

Exercise 6

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 7

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 8

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

Recursion

19 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 9

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 10

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`

Exercise 11

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 10 to implement this.

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

Exercise 12

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

20 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.

21 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
```

(More about strings later.)

22 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

23 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
}
```

24 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 13

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;
```

25 Life time vs reachability

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

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

Lambdas

26 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);
```

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 unambiguous;
- Definition: function body.

27 Assign lambda to variable

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

(For technical reasons it is necessary to use `auto` here. Otherwise this is a typed variable declaration as you have seen before.)

Exercise 14

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`

Lambdas as parameter

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

Instead:

```
#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 15

Rewrite the Newton exercise above to use a function with this prototype. Call the function directly with the lambda functions as arguments, that is, without assigning them to variables.

28 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
6:7776
7:16807
8:32768
```

Exercise 16

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 f( 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 17

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 root finder you coded earlier.

29 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
```