Functions

Victor Eijkhout, Susan Lindsey

Fall 2024
last formatted: September 1, 2024
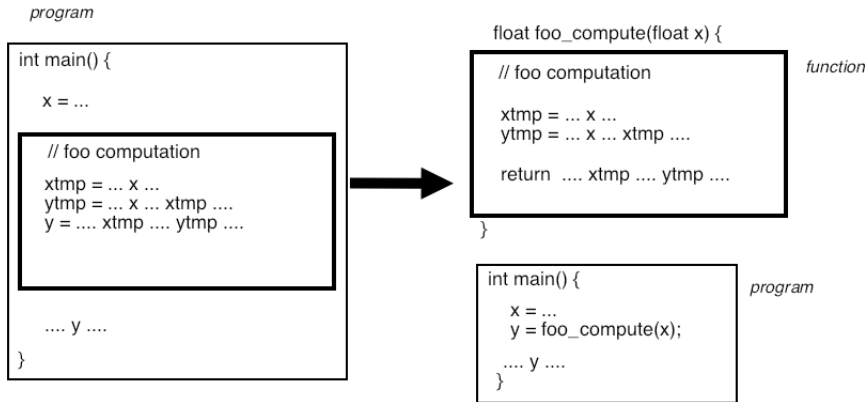
Function basics

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.

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

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

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

```
Code:

  // func/bisect1.cpp
  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;
```

```
Output:

Zero happens at:
    ↪1.4375
```

4. **Introducing functions, step 1**

Introduce a function for the
expression m*m*m – m*m–1:

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

Used in main:

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

5. Introducing functions, step 2

Function:

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

New main:

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

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

Exercise 1

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

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

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

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

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)

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:

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

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

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

This is purely a matter of style.

10. **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.

11. **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`.)

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.

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

Programming Project Exercise 3

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

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

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

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 4

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

- Read an integer *how_many* from the input, indicating how many (successive) prime numbers should be printed.

- Print that many successive primes, each on a separate line.

- (Hint: keep a variable *number_of_primes_found* that is increased whenever a new prime is found.)

Turn it in!

- If you have compiled your program, do:

  `coe_primes yourprogram.cc`

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

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

  `coe_primes -s yourprogram.cc`

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

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

  `coe_primes -i yourprogram.cc`

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

12. Background: square roots by Newton's

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

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

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

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

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

Optional exercise 5

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

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:

  // func/passvalue.cpp
  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:

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.

15. Reference

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

```
Code:
  // basic/ref.cpp
  int i;
  int &ri = i;
  i = 5;
  cout << i << "," << ri <<
      '\n';
  i *= 2;
  cout << i << "," << ri <<
      '\n';
  ri -= 3;
  cout << i << "," << ri <<
      '\n';
```

```
Output:
5,5
10,10
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  }
```

Reference syntax is cleaner than C 'pass by reference'

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:

  // basic/setbyref.cpp
  void f( int &i ) {
    i = 5;
  }
  int main() {

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

```
Output:

5
```

Compare the difference with leaving out the reference.

19. Pass by reference example 2

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

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

Exercise 6

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

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

```
Output:
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:

```
// func/divisible.cpp
cout << number;
if (is_divisible

    (number,divisor,remainder))
  cout << " is divisible by ";
else
  cout << " has remainder "
      << remainder << " from
    ";
cout << divisor << '\n';
```

Output:

```
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:

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

```
Output:

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

Recursion

20. Recursion

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

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

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:

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

```
Output:

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

Extend this idea to define powers as repeated multiplication.

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

Exercise 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 if $n == 1$:
  x
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

21. 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.

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
```

23. 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

24. 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
}
```

25. 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?

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

26. Life time vs reachability

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

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