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.

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.

2. Declaration vs definition

Function declaration:

- Function name
- Return type and types of parameters
- qualifiers (see later)

Function definition

- Declaration, plus
- parameter names and function body

3. Declaration first, definition last

Some people like the following style of defining a function:

```
// 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.

4. Background: square roots by Newton's m

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 1

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

5. 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’

6. 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.)

7. 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
}
```

Exercise 2

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 3

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 4

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

8. 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 5

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 6

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 7

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 6 to implement this.

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

Exercise 8

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

9. 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.

10. 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
```

11. 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

12. 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
}
```

13. 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 9

What is the output of this code?

```
// 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';
```

14. Life time vs reachability

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

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