# Functions

Kevin Schmidt, Susan Lindsey, Charlie Dey

Spring 2019

# Function basics

# Turn blocks of code into functions

- Code fragment with clear function:
- Turn into *subprogram*: function *definition*.
- Use by single line: function *call*.

# Function definition and call

```
for (int i=0; i<N; i++) {
  cout << i;
  if (i%2==0)
    cout << " is even";
  else
    cout << " is odd";
  cout << endl;
}
```

```
void report_evenness(int n) {
  cout << n;
  if (n%2==0)
    cout << " is even";
  else
    cout << " is odd";
  cout << endl;
}
...
int main() {
  ...
  for (int i=0; i<N; i++)
    report_evenness(i);
}
```

Code becomes more readable (though not necessarily shorter): introduce application terminology.

# Function definition and call

**Code:**

```
int double_this(int n) {
  int twice_the_input = 2*n;
  return twice_the_input;
}
  /* ... */
  int number = 3;
  cout << "Twice three is: "
       << double_this(number) << endl;
```

**Output**
**[func] twicein:**

```
Twice three is: 6
```

# Why functions?

- Easier to read
- Shorter code: reuse
- Cleaner code: local variables are no longer in the main program.
- Maintainance and debugging

# 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 fuction this can be replaced by:

```
double computation(double in) {
  return .... computation from 'in' ....
}

y = computation(x);
w = computation(v);
```

# 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:

```
int OneNorm( vector<float> a )
    {
  float s = 0;
  for (int i=0; i<n.size(); i
    ++)
    s += abs(a[i]);
  return s;
}
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.
- Using functions makes your code easier to read.
- Functions have to be defined before you can use them.

# Anatomy of a function definition

- 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.

# 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`.)

# Functions without input, without return result

```
void print_header() {
  cout << "****************" << endl;
  cout << "* Output       *" << endl;
  cout << "****************" << endl;
  }
int main() {
  print_header();
  cout << "The results for day 25:" << endl;
  // code that prints results ....
  return 0;
}
```

# Functions with input

```
void print_result(int day,float value) {
  cout << "****************" << endl;
  cout << "* Output       *" << endl;
  cout << "****************" << endl;
  cout << "The results for day " << day << ":" << endl;
  cout << "    " << value << endl;
  }
int main() {
  print_result(25,3.456);
  return 0;
}
```

# Functions with return result

```
#include <cmath>
double pi() {
  return 4*atan(1.0);
}
```

The `atan` is a *standard function*

# Review 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 1

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 2

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 3

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:

  `sdstestprime yourprogram.cc`

  where '`yourprogram.cc`' stands for the name of your source file.
- Is it reporting that your program is correct? If so, do:

  `sdstestprime -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

  `sdstestprime -i yourprogram.cc`

# Background Square roots through Newton

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

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

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

# Exercise 4

- 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

# 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'

# Pass by value example

Note that the function alters its parameter `x`:

**Code:**

```
double squared( double x ) {
  x = x*x;
  return x;
}
  /* ... */
  number = 5.1;
  cout << "Input starts as: "
       << number << endl;
  other = squared(number);
  cout << "Input var is now: "
       << number << endl;
  cout << "Output var is: "
       << other << endl;
```

**Output**
**[func] passvalue:**

```
Input starts as: 5.1
Input var is now: 5.1
Output var is: 26.01
```

but the argument in the main program is not affected.

# 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.)

# 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
}
```

# 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 also called ‘input’, ‘output’, ‘throughput’.

# 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.

# Pass by reference example 2

```
bool can_read_value( int &value ) {
  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;
```

# Exercise 5

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

```
int i=2,j=3;
swapij(i,j);
// now i==3 and j==2
```

# Exercise 6

Write a `bool` function that tests divisibility and returns a remainder:

```
int number,divisor,remainder;
// read in the number and divisor
if ( is_divisible(number,divisor,remainder) )
  cout << number << " is divisible by " << divisor << endl;
else
  cout << number << "/" << divisor <<
      " has remainder " << remainder << endl;
```

# Exercise 7

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

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

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: (0.707107,0.70
Rotated to the y-axis: (0,1)
```

# Recursion

# Recursion

A functions is allowed to call itself, making it a *recursive function*. For example, 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 8

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.

# Exercise 9

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

# 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.

# Polymorphic functions

You can have multiple functions with the same name:

```
double sum(double a,double b) {
  return a+b; }
double sum(double a,double b,double c) {
  return a+b+c; }
```

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

# Scope

# 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
}
```

# 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 10

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;
```

# Life time vs reachability

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

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