# Functions

Victor Eijkhout, Harika Gurram,
Je'aime Powell, Charley Dey

Fall 2018

# Function basics

# Turn blocks of code into functions

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

# Example

The code for an odd/even test

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

becomes

```
void report_evenness(int n) {
  cout << i;
  if (i%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.

# Code reuse

Example: multiple norm calculations:

Repeated code:

```
float s = 0;
for (int i=0; i<nx; i++)
  s += abs(x[i]);
cout << "Inf norm x: " << s << endl;
s = 0;
for (int i=0; i<ny; i++)
  s += abs(y[i]);
cout << "Inf norm y: " << s << endl;
```

becomes:

```
int InfNorm( float a[],int n) {
  float s = 0;
  for (int i=0; i<n; i++)
    s += abs(a[i]);
  return s;
}
int main() {
  ... // stuff
  cout << "Inf norm x: "
       << InfNorm(x,nx) << endl;
  cout << "Inf norm y: "
       << InfNorm(y,ny) << endl;
```

Code becomes shorter, easier to maintain.
(Don't worry about array stuff in this example)
Introduces application terminology.

# 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);
}
```

# Program with function

**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 from running twicein in code directory func:**

```
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

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

can be replaced by

```
double computation(double in) {
  return .... computation from ‘in’ ....
}

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

# 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.

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

# Functions with return result

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

The `atan` is a *standard function*

# Exercise 1

```
class Point {
private:
  float x,y;
public:
  Point(float ux,float uy) { x = ux; y = uy; };
  float distance(Point other) {
    float xd = x-other.x, yd = y-other.y;
    return sqrt( xd*xd + yd*yd );
  };
};
```

# Project Exercise 2

```
bool isprime(int number) {
  for (int divisor=2; divisor<number; divisor++) {
    if (number%divisor==0) {
      return false;
    }
  }
  return true;
}
```

# Project Exercise 3

Take your prime number testing function `is_prime`, and use it to write 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.)

# Exercise 4

```
double func( double x,double number ) {
  return x*x-number;
}
double deriv( double x,double number ) {
  return 2*x;
}

double newton_root( double number ) {
  double guess = .5, prev = 0;
  while (abs(func(guess,number))>1.e-5) {
    prev = guess;
    guess = prev - func(prev,number) / deriv(prev,number);
    cout << ".. current guess: " << guess << endl;
  }
  return guess;
}
```

# Parameter passing

# Mathematical type function

Pretty good design:

- pass data into a function,
- return result through `return` statement.
- Parameters are copied into the function.
- *pass by value*
- 'functional programming'

# Functional programming example

**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 from running passvalue in code directory func:**

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

# 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 from running ref in code directory basic:**

```
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 from running setbyref in code directory basic:**

```
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 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 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

```
class Point {
public:
  float x,y;
public:
  Point() { x = NAN; y = NAN; };
  Point(float ux,float uy) { x = ux; y = uy; };
  float distance(Point other) {
    float xd = x-other.x, yd = y-other.y;
    return sqrt( xd*xd + yd*yd );
  };
  /* ... */
};
```

# Recursion

# Recursion

Functions are allowed to call themselves, which is known as *recursion*. 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

```
int sum_of_squares( int ton ) {
  if (ton<=0)
    return 0;
  else return ton*ton+sum_of_squares(ton-1);
};
```

# 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.

# Shadowing and scope are lexical

This is independent of dynamic / runtime behaviour!

**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;
```

**Output from running shadowfalse in code directory basic:**

```
Global: 3
Global again: 3
```

# Life time vs reachability

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

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