### **Functions**

Victor Eijkhout and Charlie Dey

spring 2017



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



# Why functions?

- Easier to read
- Shorter code: reuse
- Maintainance and debugging



### **Prime function**

Using a function, primality testing would look like:

```
bool isprime;
number = 13;
isprime = prime_test_function(number);
```



# Anatomy of a function definition

- Result type: what's computed. void if no result
- Name: make it descriptive.
- Arguments: zero or more. int i,double x,double y
- Body: any length.
- Return statement: usually at the end, but can be anywhere; the computed result.



# **Program with function**

```
#input <iostream>
using namespace std;
int twice_function(int n) {
  int twice_the_input = 2*n;
  return twice_the_input;
int main() {
  int number = 3;
  cout << "Twice three is: " <<
    twice_function(number) << endl;</pre>
  return 0;
```



### **Function call**

#### The function call

- 1. causes the function body to be executed, and
- 2. the function call is replaced by whatever you return.
- 3. (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 << "* Ouput
                     *" << endl;
  cout << "********** << endl:
int main() {
 print_header();
  cout << "The results for day 25:" << endl;</pre>
  // code that prints results ....
 return 0;
```



# **Functions with input**

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



## Functions with return result

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



# Scope

Function body is a scope: local variables.

No local functions.



# Project Exercise 1

Write a function that takes an integer input, and return a boolean corresponding to whether the input was prime.

```
bool isprime;
isprime = prime_test_function(13);
```

Read the number in, and print the value of the boolean.



# **Project Exercise 2**

Take the prime number testing program, and modify it to read in how many prime numbers you want to print. Print that many successive primes. Keep a variable number\_of\_primes\_found that is increased whenever a new prime is found.



# 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



### Exercise 3

Early computers had no hardware for computing a square root. Instead, they used *Newton's method*. Suppose you want to compute

$$x = \sqrt{y}$$
.

This is equivalent to finding the zero of

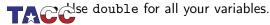
$$f(x) = x^2 - y.$$

Newton's method does this by evaluating

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

until the guess is accurate enough.

- Write functions f(x,y) and deriv(x,y), and a function newton\_root that uses f and deriv to iterate to some precision.
- Take an initial guess for x, not zero.
- As a stopping test, use  $|f(x,y)| < 10^{-5}$ .



# Results other than through return

#### Also good design:

- Return no function result,
- or return (0 is success, nonzero various informative statuses),
   and
- return other information by changing the parameters.
- pass by reference



# Pass by reference example

```
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;
if (!can read value(n))
  // if you can't read the value, set a default
  n = 10;
```



### **Exercise 4**

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

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



### Exercise 5

Write a function that tests divisibility and returns a remainder:



### 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} \text{int factorial( int n ) } \{ \text{if (n==1)} \text{return 1;} \text{else} \text{return n*factorial(n-1);} \}
```



### Exercise 6

The sum of squares:

$$S_n = \sum_{n=1}^N n^2$$

can be defined recursively as

$$S_1 = 1,$$
  $S_n = n^2 + S_{n-1}.$ 

Write a recursive function that implements this second definition. Test it on numbers that are input by the user.

Then write a program that prints the first 100 sums of squares.

### Exercise 7

Write a recursive function for computing Fibonacci numbers:

$$F_0 = 1,$$
  $F_1 = 1,$   $F_n = F_{n-1} + F_{n-2}$ 

First write a program that computes  $F_n$  for a value n that is input by the user.

Then write a program that prints out a sequence of Fibonacci numbers; the user should input how many.

# **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 input parameters: can not differ only in return type.



# **Default arguments**

Functions can have *default argument*(s):

```
double distance( double x, double y=0. ) {
  return sqrt( (x-y)*(x-y) );
}
```

Any default argument(s) should come last in the parameter list.



# **Prototypes**



### Forward declaration

#### Problem:

```
int g(int i) { return f(i); }
int f(int i) { return g(i); }
```

leads to compiler error.

Use prototype for *forward declaration*:

```
int f(int); // prototype of f
int g(int i) { return f(i); }
int f(int i) { return g(i); }
```

