Programming seminar 2015
Session 4: Functions and Pointers

Massimo Cavallaro

School of Mathematical Sciences
Queen Mary, University of London

11 March 2015

Outline

Outline

Motivation

Real-world computer programs are (much) larger than those seen so far. The strategy to develop, maintain and understand large programs is to use small components. A *function* is one of these components. It is a block of code that typically takes parameters and returns a value.

Prototypes

Example: Math library functions

You are already familiar with some functions, e.g.,

```
exp(x)
cos(x)
pow(x)
fabs(x)
floor(x)
ceil(x)
fmod(x)
```

These functions perform a precise task and can be inserted in any part of the program. To use them you need:

```
#include <cmath>
```

But why?

Prototypes

Every function — whether one we've written ourselves or one we're using from somewhere else — must have a prototype before it can be called. A prototype describes the type of the value (if any) returned by the function, and the types of all its parameters. Here are some example prototypes:

```
double sin(double angle);
double pow(double base, double ↩
    exponent);
void my_function(int a, char b, ↩
    string c);
```

The prototype should appear at the top level, and before the function is called.

Prototypes

If we are using a library function, we typically get its prototype by including the appropriate **header file** for the library (this is what, e.g., `#include <cmath>` does: it takes the contents of the system header file `cmath`, in which are the prototypes for the standard mathematical functions, and adds it to the current file).

C++ Standard Library header files

Not only mathematical functions...

The following header files contain the prototypes for some of the functions you used in the last tutorials, They contain also definitions of types and constants need by those functions:

- `<iostream>` def. the standard input/output stream objects (`std::cout`, `std::cin`)
- `<cstdlib>` def. the random number generator (`rand()`)
- `<cstdio>` def. the c-style input/output functions (`printf(const char *format, ...)`)
- `<ctime>` Contains func. prototypes and types for manipulating the time and date.
- `<cstdarg>` Support for the variadic function (last slide).

For a complete list see, e.g.,
http://www.cplusplus.com/reference/

Calling functions

We call a function by writing its name, followed by brackets containing the (actual) parameters (i.e. the values the parameters should take), separated by commas; if there are no parameters, the brackets are left empty. For example:

```
double some_sine = sin(100);
printf("Hello, world!\n");
int random_num = rand();
```

See `functions.cpp`

Outline

Writing our own functions

The first line of a function is the same as its prototype, except that it doesn't end in a semicolon and must contain the argument names. The **body** of the function follows — that is, the code that is to be executed when the function is called. The `return` statement is used to end the function and return a value.

```
int some_function(int n){
  if(n <= 0) return 0;
  cout << "Input was positive." << ↩
     endl;
  return 100 * n;
}
```

See `myfirstfunction.cpp`.

Variables in functions: scope

Variables declared inside a function are said to be **local** to that function — that is, they're visible within that function only. So this is an error:

```
int main() {
  int i;
  cout << fn() << endl;
  return 0;
}
int fn() {
  return i * 2; // ERROR
}
```

To make a variable visible to all functions — we say, **global** — define it at the top level, outside any function. But avoid this if possible: use parameters instead. See `glocal.cpp`

Parameter-passing in detail

What is the output when running the following?

```
int main() {
  int i = 10;
  mystery(i);
  cout << "i is " << i << endl;
  return 0;
}
void mystery(int n) {
  n++;
  cout << "n is " << n << endl;
}
```

Copy/Paste this snippet and compile. Hint: It's nothing to do with the names of the variables!

Parameters are passed by value

When we call a function, *copies of the values* of the actual parameters are made for use by the function, so changes to the parameters within the function have no effect on the caller. This arrangement is known as **passing by value**.
We can do more.
But first...

Exercises!

based on the last tutorial!

- Write a function that prints n times `hello world`
- Write a function that calculates and returns the factorial of n. See `forloop3factorial.cpp`
- Write a function `mypow(x,n)` that calculates and returns x^n, $n \in \mathbb{N}$. Compare its speed with those of `pow(x,n)`.
- Write a function that calculates and returns the exponential of x with a certain precision *prec*. Make use of `mypow(x,n)`. See `whileloopexponential.cpp`

Outline

Pass by reference

To get the opposite behaviour, whereby a copy is *not* made, and the function operates on an actual parameter directly, so that changes to that parameter *do* affect the caller, specify that that parameter is a **reference**, which is achieved by adding an ampersand to the end of the type (both in definition and prototype), e.g.

```
void mystery(int& n){
  n++;
  cout << "n is " << n << endl;
}
```

This is called **passing by reference**. How does it works?

Pass by reference 2

Example

Write a function that double the value in the caller, and returns void. see `ref.cpp`

Roughly speaking, the "ampersand" `&` means "address of". In the previous snippet, the address of the variable `n` is copied and passed. That address can't change. But, after, the function works directly on the memory with address `&n`.

Pass by reference 3

ANSI C is syntactically more coherent

You can't pass by reference and you must use the *dereference operator* * in order to pass an address.

```
#include <stdio.h>
void modify(int* c){
    *c = 6;
}
int main(){
    int d = 10;
    modify(&d);
    printf("%d\n",d);
    return 1;
}
```

In function `modify`, `c` is a pointer to an integer while `*c` is an integer. See `pointer.c`

Outline

Variable number of arguments

If we include the header `cstdarg`, we have access to some useful functions that allow us to define function with a variable number of arguments. See `variadic.cpp`.

```
double ArithmeticMean(int num,...){
    va_list valist; //declaration
    double sum = 0.0;
    int i;
    va_start(valist, num); //↩
        initialization of valist
    for (i = 0; i < num; i++)
        sum += va_arg(valist, int); ↩
            //access
    va_end(valist);
    return sum/num;
}
```