Chapter 6

Functions

Objectives:

- Subprograms in C
- Writing and calling a function
- Local and global variables
- Standard Math Functions

6. Subprograms and C Functions

- A large program is easier to code, manage and maintain if it is written in a modular fashion.
  - Modular program design means that specific tasks are isolated and coded into separate modules or subprograms.
  - In C, a module or subprogram is called a function.
- Modular program design using functions offers many advantages since functions:
  - allow the abstraction of code during program design and development,
  - allow the re-use of code elsewhere in the program or in another program,
  - avoid the repetition of code in the same program or in another program.

6.1 The Subprogram

- The subprogram is composed of a name, parameters and an instruction bloc; and it may return a value
  - The parameters are variables used to receive values from the caller

Program Memory

```
poly(x, a0, a1, a2, a3)
    Assign a0*x*x*x to fx
    Assign fx+(a1*x*x) to fx
    Assign fx+(a2*x)+a3 to fx
    Return fx
```

Working Memory

x	
a0	
a1	
a2	
a3	
fx	

CPU

Calling a subprogram

Program Memory

```
find_root(left, right, a0, a1, a2, a3)
    Assign poly(left, a0, a1, a2, a3) to func_left
    Assign poly(right, a0, a1, a2, a3) à func_right
    If func_left X func_right is smaller than ....
        If func_left is smaller than ....
            Print "A root exists at ", func_left
    Otherwise
        If func_left X func_right is smaller than 0
            Print "A root exists at ", (right+left)/2

poly(x, a0, a1, a3, a3)
    Assign a0*x*x*x to fx
    Assign fx+(a1*x*x) to fx
    Assign fx+(a2*x)+a3 to fx
    Return fx
```

CPU

UNIVERSITÉ D'OTTAWA • UNIVERSITY OF OTTAWA
Génie • Engineering

Subprogram operation

- A piece of working memory is reserved for and during the execution of a subprogram
- The parameters and variables are created in the reserved working memory when the subprogram is executed
- Note that the same name for variables can be used in different subprograms
  - Although the names are the same, the variables are different – why?
  - The variables (and parameters) are **local** to the subprogram.
  - They are in reality associated to different addresses in memory.
- When the execution of the subprogram is completed, the variables (and parameters) no longer exist and the working memory can be re-used (e.g. for the execution of another subprogram)

Calling a subprogram

- A call to a subprogram can be made within an instruction using the subprogram name and a set of arguments
  - When a subprogram is called, the execution of the calling subprogram is suspended
  - The arguments are expressions that are evaluated to give values
  - The argument values are stored in the parameters of the subprogram to be executed

    The order of the arguments correspond to the order of the parameters
  - The subprogram can return values to the calling subprogram (or calling main program)

    This means that a subprogram call can be used in any expression just like a variable

    E.g. ***Assign 3+poly(4.2, 3.2, -2.5, 1, 8.8) to z***

The main program

Program Memory

Read values into a0, a1, a2, a3
Read values into begin, end
Read a value into subinter
Assign (end-begin)/subinter to n
Assign 0 to ctr
Repeat while ctr is not equal to n
    Assign start+(ctr*subinter) to ak
    Assign start+((ctr+1)*subinter) to bk
    If bk est larger than end
        Assign end to bk
    find_root(ak, bk, a0, a1, a2, a3)
    Increment ctr
find_root(fin, fin, droite, a0, a1, a2, a3)

Working Memory

begin
end
a0
a1
a2
a3
subinter
n
ctr
ak
bk

CPU

- The main program operates in the same fashion as a subprogram

UNIVERSITÉ D'OTTAWA • UNIVERSITY OF OTTAWA
Génie • Engineering

Global Variables

Program Memory

Define glob_int as global
Assign 2 to main_int
Assign 2 to glob_int
print_ints(main_int)
Print “main_int after call: ”, main_int
Print “glob_ing after call: “, glob_int

print_ints(par_int)
Assign 2 to loc_int
Print “Local variable: “, loc_int
Print “Parameter: “, par_int
Increment par_int
Print “Global variable: “, glob_int
Increment glob_int

Working Memory

main_int

loc_int

par_int

CPU

Global Memory

glob_int

Global Variables

- Defining a global variable places variables in global memory
  - Such variables are not associated to any single subprogram and exist during the execution of the program
  - Global variables are visible to all subprograms
- Use of global variables
  - Extensive use of global variables can lead to programs difficult to understand, debug and maintain.
  - For example, it is easier to understand a subprogram when it receives values of the caller’s local variables in parameters instead of using global variables. It also makes it easier to re-use the subprogram.
- In this course do not use global variables unless directed to do so.

6.2 The C Function

- In C, the subprogram is called a function.

- In C, a function is invoked via a function call. A function call begins with the name of the desired function followed by an argument list in parentheses. The argument list provides the data to be passed to the function.

  E.g. `printf("Hello World.\n");`

  – In this example, the function `printf` is invoked through its name and the argument list contains only one argument to be passed which is the character string:

  `"Hello World.\n"`

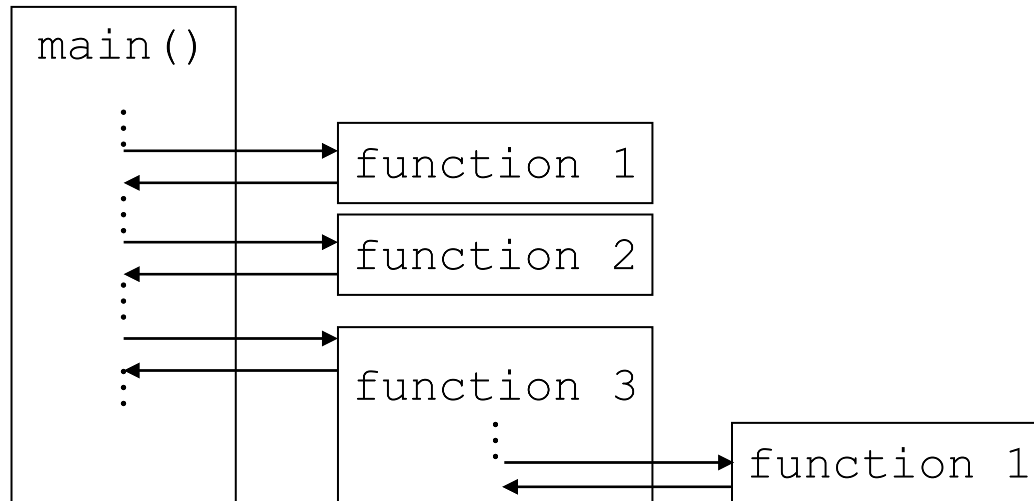
Invoking (Calling) Functions

- The function, once invoked, executes the desired task with the given arguments and returns control to the caller when done.

E.g.:
```
#include <stdio.h>
void main()
{
    int num;
    num = 1;
    printf("Value of num: %d\n", num);
    num = 10;
}
```

- In this example, `main` is the caller of the function `printf`.
- Execution in `main` begins with the assignment of `1` to variable `num`
- The function `printf` is then invoked with the arguments to be passed listed between parentheses.
- Execution is now within the function `printf` which prints out the arguments to the screen.
- Once `printf` has completed its task, program control is returned to `main`.
- Execution continues with the last command in `main`, which is the assignment of `10` to the variable `num`

- Execution flow with many function calls:



- Functions can also return a value back to the caller.

6.3 Standard Functions in C

- There exists in C a standard library of functions which are available for general use.
  - Avoid re-inventing the wheel! Familiarize yourselves with the C standard functions and use them. The standard functions are defined by ANSI (American National Standards Institute) and are available with any C compiler that adheres to the ANSI C standard.
  - We find in the standard library: mathematical functions, input/output functions, file manipulation functions and functions to manipulate character strings, among others.
  - The course textbook (Appendix A) lists and describes some of the C standard functions
    - `printf` and `scanf` are ANSI C standard functions.

Standard Math Functions

- The course textbook contains a list and description of standard C math functions (see appendix A).
  - The most popular math functions are available.
- The preprocessor directive `#include <math.h>` provides the C compiler with definitions used by the standard math functions in the program. This directive is normally placed after the directive `#include <stdio.h>` before `main()`.
- All standard math functions except three (`frexp`, `ldexp` and `modf`) require arguments of type `double` and all standard math functions return a number of type `double`.
- The value returned by a math function can be
  - assigned to a variable for future use:

```
x = sin(angle);
```

  - used in an arithmetic expression, or as an argument sent in to another function:

```
c = sqrt(PI + pow(z, 3.0) );
```

- An argument to a math function can be a symbolic constant, a variable or a mathematical expression.

- The rules of promotion are applied to the arguments of math functions if they are not of type `double`. These same rules are applied to the value returned by the function if it is not stored into a variable of type `double`.

- Some of the most commonly encountered math functions:

Math Function	Standard C Math Function	Note on Usage
$\sqrt{x}$	`sqrt(x)`	`x` must be ≥ 0.0
x^y	`pow(x,y)`	
e^x	`exp(x)`	
$\log_e x$	`log(x)`	
$\log_{10} x$	`log10(x)`	
$\|x\|$	`fabs(x)`	
sin(x)	`sin(x)`	`x` is in radians.
cos(x)	`cos(x)`	`x` is in radians.
tan(x)	`tan(x)`	`x` is in radians.
arcsin(x)	`asin(x)`	$-1 \leq x \leq 1$ and $-\pi/2 \leq return \leq \pi/2$, in radians.
arccos(x)	`acos(x)`	$-1 \leq x \leq 1$ and $0 \leq return \leq \pi$, in radians.
arctan(x)	`atan(x)`	$-\pi/2 \leq return \leq \pi/2$, in radians.
sinh(x)	`sinh(x)`	Hyperbolic functions.
cosh(x)	`cosh(x)`	Hyperbolic functions.
tanh(x)	`tanh(x)`	Hyperbolic functions.

- The function `ceil(x)` rounds `x` to the smallest integer that is greater than or equal to `x`.
  - E.g.:
    - `ceil(9.0)` yields 9.0
    - `ceil(9.2)` yields 10.0
    - `ceil(9.5)` yields 10.0
    - `ceil(9.8)` yields 10.0
    - `ceil(-9.8)` yields -9.0
    - `ceil(-9.5)` yields -9.0
    - `ceil(-9.2)` yields -9.0
    - `ceil(-9.0)` yields -9.0

    Returns an integer as a `double`.

- The function `floor(x)` rounds `x` to the largest integer that is less than or equal to `x`.
  - E.g.:
    - `floor(9.0)` yields 9.0
    - `floor(9.2)` yields 9.0
    - `floor(9.5)` yields 9.0
    - `floor(9.8)` yields 9.0
    - `floor(-9.8)` yields -10.0
    - `floor(-9.5)` yields -10.0
    - `floor(-9.2)` yields -10.0
    - `floor(-9.0)` yields -9.0

    Returns an integer as a `double`.

Generation of Random Numbers

- Random numbers are often used to simulate a random process, such as a coin toss or a dice roll.
- The standard functions `rand()` and `srand()` can be used to generate a pseudo-random sequence of numbers. These sequences are called pseudo-random since the numbers eventually start repeating themselves.
- The function `rand()` is of type `int`, thus it returns an integer, and it does not require an argument.
  - The integer returned by `rand()` is contained between 0 and `RAND_MAX` where `RAND_MAX` is a symbolic constant.
  - The value associated with `RAND_MAX`, defined in `stdlib.h` (`#include <stdlib.h>`), is usually 32767.
- From one execution to another, `rand()` generates the same sequence of pseudo-random numbers. This sequence can be changed via the function `srand()` which changes the root or the seed of the random number generator.
  - The function `srand()` does not return a value.
  - The function `srand()` requires one argument which is a positive integer of type `unsigned int` (default value of the seed is 1).

Sequences of Random Integers

- We often wish to generate a sequence of random integers that are within a prescribed range. The modulus operator `%` is quite useful for this task.

E.g.:

```
int integer;
integer = rand() % 8;
integer = 1 + rand() % 6;
integer = rand() % 51 - 25;
```

→ $0 \le \text{integer} \le 7$

→ $1 \le \text{integer} \le 6$

→ $-25 \le \text{integer} \le 25$

- The following general formula can be used to generate random integers within the range $a \le \text{integer} \le b$:

```
int a, b, integer;
integer = a + rand() % (b - a + 1);
```

  – The expression `(b - a + 1)` specifies the width of the range and `a` specifies its beginning.

Sequences of Random Real Numbers

- We often wish to generate a sequence of random real numbers that are within a prescribed range.
  - The following bit of code will produce a random real number within the range: $x \le real_no \le y$.

```
float x, y, real_no;
    :
/* Random real number in the range 0.0 to 1.0. */
real_no = (float) rand() / RAND_MAX;
/* Scale to reduce the range to 0.0 to y - x. */
real_no = real_no * (y-x);
/* Add x to displace the range to x to y. */
real_no = real_no + x;
```

- Note that we may also work with numbers of type `double`.

6.4 Definition of Functions

- It is possible to program your own functions in C; this is called function definition. The syntax of a function definition in C is:

```
type function_name(parameter list)
{
    C declarations
    C commands
}
```

Careful: there is no ; here.

  – The function header describes the communication channel that will be established with the caller.
  – The name of the function is created from the same symbols as a variable name.
  – The type of the function identifies the type of the value returned by the function.
    - If type is `void` then the function returns nothing to the caller.
    - If type is omitted then the compiler assumes that the function returns an `int`.

Function Parameters and Variables

- The parameter list contains the declaration of the variables to receive arguments passed to the function.
  - Individual declarations are separated by `,` (commas).
  - If the function does not receive arguments then the parameter list is of type `void`.
  - If the type of a parameter is omitted then the compiler assumes that it is an `int`.
- All variable declarations and instructions in the function are placed between `{}`, i.e. an instruction bloc.
  - All variables declared in a function are local; this means that they are not available and are not known outside of the function where they are defined.

Returning from a Function

- There are 3 ways of returning program control from a function back to the caller:
  1. Reaching the end of the function as delimited by the }
  2. By executing the command: `return;`
  3. By executing the command: `return expression;`
  - Mechanisms 1 or 2 are used in a function that does not return a value to the caller; **i.e: functions of type `void`.**
  - Mechanism 3 must be used in a function that returns a value to the caller; `expression` must evaluate to a value having the same type as the function declaration.
- A function cannot be defined within another function.

Example

- Write a function that accepts three float numbers and returns the average
- Write the main() function that prompts the user for 3 numbers, call the function, receive the result, and prints the result on the output screen.

- Example: a simple function that prints out Hello World! to the screen.

```
void hello(void)
{
    printf("Hello World!\n");
}
```

This function receives no arguments and returns nothing to the caller.

- Example: a simple function that prints out the largest of 2 integers to the screen.

```
void prtmax(int a, int b)
{
    if (a > b)
        printf("%d is the largest integer.\n", a);
    else
        printf("%d is the largest integer.\n", b);
}
```

This function receives 2 arguments (integers) and returns nothing to the caller.

- Example: a simple function that returns the largest of 2 integers to the caller.

```
int max(int a, int b)
{
    if (a > b)
        return a;
    else
        return b;
}
```

This function receives 2 arguments (integers) and returns an integer to the caller.

- Example: a simple function that returns the largest of 2 integers to the caller.
  - Performs the same task as the previous example but in better style.

```
int max(int a, int b)
{
   int largest;

   if (a > b)
      largest = a;
   else
      largest = b;

   return largest;
}
```

This function receives 2 arguments (integers) and returns an integer to the caller.

Function Prototypes

- The syntax of a function prototype is:

```
type function_name(type, type, … , type);
```

- The prototype:
  - declares that `main()` will call the function: `function_name`,
  - defines the type of the value returned by the function,
  - identifies the number of parameters and the order of the types passed to the function.
- The prototype of a function must have:
  - the same name as in the function definition,
  - the same function type as in the definition,
  - the same number of parameters as in the definition, and
  - the same order of parameter types.
  - The parameter names are optional.
- The function prototypes are usually listed after the preprocessor directives but before `void main()` in the program file.

- The prototypes for our simple example functions are:

```
void hello(void);
void prtmax(int, int);
int max(int, int);
```

- Heading files such as `stdio.h`, `math.h` and `stdlib.h` contain among other things, function prototypes for the standard C functions.

- Function prototypes are not required in a C program but you are <u>strongly encouraged</u> to include them for all functions called in the main program since they:
  - help others understand your program file,
  - can help avoid errors when calling the function.

On the Passage of Arguments to a Function

- In many high level programming languages, there exists 2 ways of passing data to the function being called:
  - Pass by value.
  - Pass by reference.
- In a pass by value, the argument is a value is passed to the function (placed in a parameter); thus any source of the value remains unchanged in the caller even if the corresponding parameter is modified in the called function. Arguments can be expressions that are evaluated to values.
- In a pass by reference, the argument is an address to a value in memory passed to the function (placed in a parameter); thus the contents of the address (e.g. variable in the calling function) will be changed in the caller if the contents are modified in the called function.

On the Passage of Arguments to a C Function

- **In C <u>most</u> arguments use pass by value**;
  - it is possible however to make a pass by reference using addressing operators (as we have seen with `scanf`) and
  - as we shall see later, arrays and strings are passed by reference.

- The rules of promotion are applied to mixed type arguments and to the type returned by the function.

Local Variables: Variables Declared in a C Function (including main())

- All variables declared in in a function within the opening `{` and closing `}` of the function's instruction block are local to the function:
  - Known to the function and not known in other functions in the program.
  - The function parameters are also local variables.

Global Variables

- All variables declared outside of function instruction blocks (usually before `main()` and usually after the function prototypes) are:
  - Known to all of the functions defined in the same C file.
- Global variables should be avoided since:
  - A global variable does not appear in the parameter list of the function definition; hence, the function definition is not as clear as it can be; i.e: the I/O's are not properly described.
  - A global variable modified "by accident" in a function will appear modified everywhere else, rendering program debugging more difficult.
- A global variable should be used only in exceptional circumstances; i.e: only when the readability of the code would be greatly improved or when the algorithm prescribes it.

Example

```
#include <stdio.h>
int x = 10;                    ← Global variable

      void main()
      {
          int x;               ← Local variable
          x = 1;
          printf(“x = %f”, x);
      }

      Void fn()
      {
          printf(“x = %f”, x);
      }
```

Thank You!

Ευχαριστώ

MULTUMESC

ขอบคุณ

Vielen Dank

Teşekkürler

Merci

DMnvwd

شكراً

متشكرم

Gracias

THANK YOU

Grazie

Bedankt

Dankie

Köszönettel

Hvala

Obrigado!

謝 謝

Asante

شكريا

Díky

GADDA GUEY

WAD MAHAD
SAN TAHAY

감사합니다

Urakoze