PROGRAMMING IN C: FUNCTIONS

COMP1206 - PROGRAMMING II

Enrico Marchioni
e.marchioni@soton.ac.uk
Building 32 - Room 4019

- How to define a function in C.
- How to use functions with no arguments / multiple arguments that return a value / no value.
- How functions can take arrays as arguments and how they can modify arrays.
- How functions can call each other and how to avoid problems of conflicting types with functions.
- How functions can have pointers as arguments and how they can return pointers.

FUNCTIONS

- A function is a routine that performs certain actions.
- A function possibly requires some arguments to perform these actions and it may (or may not) return a certain output.
- The `main` routine is an example of a function.
- It has the form

```
int main (void) { }
```

  where:
  - `main` is the name of the function;
  - `(void)` means `main` takes no argument;
  - `int` mean `main` returns an integer value.
- `printf` and `scanf` are examples of functions we have been using so far.

- In C, it is possible to define your own functions.
- These functions live outside of the `main` routine and can be called from it to perform certain computations.
- When you call a function from `main`, program execution is transferred to the function. Execution is then returned to `main` when the function has ended its computation.
- You can define a function as follows:

```
type name (type argument, type argument ...){ }
```

  - `name` is whatever you choose to call the function.
  - `type argument` is an argument, along with its type, needed for the function to perform the required computations.
  - `type` is the kind of values you expect your function to return.

```c
// FUNCTION WITH VOID ARGUMENT AND RETURN
#include <stdio.h>

void useless (void)
{
    printf("Hello, I'm a useless function!\n");
}

int main (void)
{
    useless();
    useless();
    useless();
    return 0;
}
```

```
Hello, I'm a useless function!
Hello, I'm a useless function!
Hello, I'm a useless function!
Program ended with exit code: 0
```

- `useless` takes no argument and returns no value.
- This function simply displays some text.
- Every time you call it, the same text specified in the function's definition will be displayed.

- The first C program we saw was for computing triangular numbers.
- We can define a function:

```
int calculateTriangularNumber (int n) { }
```

- We can then call this function at any point in our main rotine and assign its return value to other variables and use it:

```
output = calculateTriangularNumber(number);
```

- Alternatively, we can simply call the function to display the value directly:

```
printf("Triangular number #%i is %i\n", number,
    calculateTriangularNumber(number));
```

```c
// FUNCTION THAT COMPUTES TRIANGULAR NUMBER
#include <stdio.h>

int calculateTriangularNumber (int n)
{
    int i, triangularNumber = 0;
    for(i=1; i<=n; ++i)
        triangularNumber += i;

    return triangularNumber;
}


int main (void)
{
    int number, output;

    printf ("What triangular number do you want?\n");
    scanf ("%i", &number);

    output = calculateTriangularNumber(number);

    printf("Triangular number #%i is %i\n", number, output);

    return 0;
}
```

```
What triangular number do you want?
1000
Triangular number #1000 is 500500
Program ended with exit code: 0
```

FUNCTIONS AND ARRAYS

- You can define functions with multiple arguments.
- Some of the arguments can be arrays.
- We define a function that returns the average of any number of elements.
- The function

  ```
  float average (float values[], int n)
  ```

  takes two arguments. The array for the values of which we want to compute the average, and the number of elements in the array.
- The function computes the average and returns a float.
- We can call this function in any program where we declare and initialize any float array.
- If our float array is called `a` and the variable for the number of elements is called `size`, to call the `average` function, use:

  ```
  average(a, size)
  ```

```c
// FUNCTION THAT COMPUTES AVERAGE OF VARIABLE LENGTH ARRAY
#include <stdio.h>

float average (float values[], int n)
{
    float tot = 0, avg;
    for (int x = 0; x < n; x++)
    {
        tot = tot + values[x];
    }
    avg = tot / n;
    return avg;
}

int main()
{
    int size;

    printf("How many elements does your array have?\n");
    scanf("%i", &size);
    float a[size];

    printf("Please insert the value of your elements\n");

    for (int x=0; x < size; x++) {
        printf("The value of a[%i]:\n", x);
        scanf("%f", &a[x]);
    }

    printf("The average is %f:\n", average(a, size));
    return 0;
}
```

- The `main` routine asks you to enter the size of the array `a`:

```
printf("How many elements does your array have?\n");
scanf("%i", &size);
```

- You are asked to intialize all the elements in the array:

```
for (int x=0; x < size; x++) {
        printf("The value of a[%i]:\n", x);
        scanf("%f", &a[x]);
```

- To display the average, `main` calls the function `average`, with arguments `a` and `size`, that returns the average value:

```
printf("The average is %f:\n", average(a, size));
```

- Consider any function

```
type newFunction (type x)
```

- When a valued is passed to `newFunction` as an argument, it is automatically copied into its formal parameter `x`.
- Any change made to `x` inside the function affects the value of `x` and not the original value.
- A function cannot directly change the value of any of its arguments, it can only change copies of them...unless the arguments are arrays.
- When a function takes a whole array as an argument, it gets passed information on where the array is stored.
- Any changes made to the elements are then made to the original array.
- This applies only to entire arrays passed as arguments and not to individual elements.

```c
// FUNCTION THAT MODIFIES CONTENT OF AN ARRAY
#include <stdio.h>

void replace(int array[5])
{
    for (int x = 0; x < 5; x++) {
        array[x] = 2 * array[x];
    }
}


int main()
{
    int a[5] = { 1, 2, 3, 4, 5};

    replace(a); // Call replace function to modify array values

    for (int x = 0; x < 5; x++) {
        printf("a[%i] = %i\n", x, a[x]);
    }

    return 0;
}
```

```
a[0] = 2
a[1] = 4
a[2] = 6
a[3] = 8
a[4] = 10
Program ended with exit code: 0
```

- The previous program defines a function `replace` that takes as argument any five element integer array:

```
void replace(int array[5])
```

- `replace` simply goes through each element of the array and multiples it by 2:

```
for (int x = 0; x < 5; x++)
     array[x] = 2 * array[x];
```

- `replace` does not return any value.
- `main` defines an integer array of 5 elements

```
int a[5] = { 1, 2, 3, 4, 5};
```

- Then it calls the function `replace`, which modifies the content of the array.
- `main` then prints the element of the array to show their value is changed.

COMPOSING FUNCTIONS

- You can define programs with multiple functions that call each other.
- The order of in which functions and defined and called can cause problems.
- Example: define a program that computes the square root of any float (see next slides!)
- Define a function `absoluteValue` that, given any float, returns its absolute value.
- Define a function `squareRoot` that, given any non-negative float, returns its square root.
- Define a program where you enter any float (positive or negative). The program will call `squareRoot`, which, in turn, will call `absoluteValue` to compute the square root of a non-negative number.

- The program on the next page computes the square root of any number you want by using the Newton-Raphson Method, which uses a "guess" at the square root, up to a selected level of precision ($< \epsilon$).
- To compute the square root of x:

  Step 1. Set the precision value ϵ.

  Step 2. Set the value of `guess` to 1.

  Step 3. If $|\texttt{guess}^2 - x| < \epsilon$, go to Step 5.

  Step 4. Set the value of `guess` to $\frac{(x/\texttt{guess}+\texttt{guess})}{2}$ and return to Step 3.

  Step 5. The `guess` is the approximation of the square root

```c
#include <stdio.h>

// ABSOLUTE VALUE FUNCTION
float absoluteValue (float x)
{
    if ( x < 0 ) x = -x; return (x);
}

// SQUARE ROOT FUNCTION
float squareRoot (float x)
{
const float epsilon = .00001; float guess = 1.0;
while ( absoluteValue (guess * guess - x) >= epsilon )
guess = ( x / guess + guess ) / 2.0;
return guess;
}

// MAIN ROUTINE
int main (void)
{
    float number;
    printf("Enter any number\n");
    scanf("%f", &number);
    printf("The square root of %f is %f\n", number, squareRoot(number));
return 0;
}
```

```
Enter any number
2
The square root of 2.000000 is 1.414216
Program ended with exit code: 0
```

- Let's try now to invert the order in which `absoluteValue` and `squareRoot` are written.
- The compiler finds a conflicting types error: why?

```
1  #include <stdio.h>
2
3  // SQUARE ROOT FUNCTION
4  float squareRoot (float x)
5  {
6      const float epsilon = .00001; float guess = 1.0;
7      while ( absoluteValue (guess * guess - x) >= epsilon )   2 ⚠ Implicit declaration of function 'absoluteValue' is invalid in C99
8          guess = ( x / guess + guess ) / 2.0;
9      return guess;
10 }
11
12 // ABSOLUTE VALUE FUNCTION
13 float absoluteValue (float x)   ● Conflicting types for 'absoluteValue'
14 {
15     if ( x < 0 ) x = -x; return (x);
16 }
17
18 // MAIN ROUTINE
19 int main (void)
20 {
21     float number;
22     printf("Enter any number\n");
23     scanf("%f", &number);
24     printf("The square root of %f is %f\n", number, squareRoot(number));
25 return 0;
26 }
```

- Whenever a call to a function is made from the `main` routine, the compiler assumes that the function returns an `int` unless:

  1. The function has been defined before it's called.
  2. The value returned by the function has been declared before the function is called.

- In the original program, `absoluteValue` is defined before the function is called from within `squareRoot`.
- The compiler knows `absoluteValue` will return a `float`.
- In the above program, where `absoluteValue` is defined after `squareRoot`, then the compiler doesn't know its type and assumes it returns an `int` by default.
- This creates a conflicting types error.

```c
#include <stdio.h>

// SQUARE ROOT FUNCTION
float squareRoot (float x)
{
    float absoluteValue (float x); // DECLARE FUNCTION HERE
    const float epsilon = .00001; float guess = 1.0;
    while ( absoluteValue (guess * guess - x) >= epsilon )
        guess = ( x / guess + guess ) / 2.0;
    return guess;
}

// ABSOLUTE VALUE FUNCTION
float absoluteValue (float x)
{
    if ( x < 0 ) x = -x; return (x);
}

// MAIN ROUTINE
int main (void)
{
    float number;
    printf("Enter any number\n");
    scanf("%f", &number);
    printf("The square root of %f is %f\n", number, squareRoot(number));
    return 0;
}
```

- To avoid this problem you can declare the functions *before* they are called.

```c
#include <stdio.h>

float absoluteValue (float x); // DECLARE FUNCTIONS HERE
float squareRoot (float x);

// MAIN ROUTINE
int main (void)
{
    float number;
    printf("Enter any number\n");
    scanf("%f", &number);
    printf("The square root of %f is %f\n", number, squareRoot(number));
    return 0;
}

// SQUARE ROOT FUNCTION
float squareRoot (float x)
{
    const float epsilon = .00001; float guess = 1.0;
    while ( absoluteValue (guess * guess - x) >= epsilon )
        guess = ( x / guess + guess ) / 2.0;
    return guess;
}

// ABSOLUTE VALUE FUNCTION
float absoluteValue (float x)
{
    if ( x < 0 ) x = -x; return (x);
}
```

- You can declare your functions at the very beginning of your code.

FUNCTION AND POINTERS

- Functions can take pointers as arguments and they can also return pointers.
- An example of a function with pointers:

```
void function (int *pointer);
```

- The value of the pointer is copied into the formal parameter when the function is called.
- Any change made to the formal parameter by the function does not affect the pointer that was passed to the function.
- Although the pointer cannot be changed by the function, the data elements that the pointer references can be changed.
- The combination of pointers and functions provides a useful tool to manipulate the content of memory addresses.
- The next program shows how to use a function with pointers to swap values at memory addresses.

```c
#include <stdio.h>

// Function that swaps contents of pointers
void exchange (int *pint1, int *pint2)
{
int temp;
temp = *pint1;
*pint1 = *pint2;
*pint2 = temp;
}

int main (void)
{
int i1=-5, i2=66, *p1=&i1, *p2=&i2;

printf ("i1 = %i, i2 = %i\n", i1, i2); // Print original values

exchange (p1, p2);
printf ("i1 = %i, i2 = %i\n", i1, i2); // Print values after exchange function

exchange (&i1, &i2);
printf("i1 = %i, i2 = %i\n", i1, i2); // Print after using exchange function a second
      time

return 0;
}
```

```
i1 = -5, i2 = 66
i1 = 66, i2 = -5
i1 = -5, i2 = 66
Program ended with exit code: 0
```

- Function `exchange` takes two integer pointers as arguments

```
void exchange (int *pint1, int *pint2)
```

- Local variable `temp` is used to hold one of the values while the exchange is made.
- Value at `point1` is copied at the address of `temp`.
- Value at `point2` is copied at the address of `point1`.
- Value at `temp` is copied at the address of `point2`.
- `main` routine defines pointers `p1` and `p2` that point at two integers.
- `main` calls `exchange` to swap the values at the memory address of `p1` and `p2`

- You can also define functions that return a pointer
- You can do so as follows:

```
int *newFunction (int x, int y, ...)
```

- In the next program we define the function `greatest` that returns a pointer to the greatest of three arguments:

```
int *greatest (int a, int b, int c)
```

- Given `a`, `b`, `c`, the function finds the maximum and assigns its memory address to a pointer `g`:

```
g = *a;
```

- `greatest` returns the pointer `g`:

```
return g;
```

```c
// FUNCTION THAT RETURNS POINTER
#include <stdio.h>

int *greatest(int a, int b, int c)
{
    int *g;
    if(a > b && a > c) g = &a;
    else if(b > a && b > c) g = &b;
    else g = &c;
    return g;
}

int main() {
    int *p;
    p = greatest(5, 25, 16);
    printf("The greatest value is %d\n",*p);
    return(0);
}
```

```
The greatest value is 25
Program ended with exit code: 0
```