Introduction to C

Tutorial 8: Functions
Agenda

- Functions
- Call stack
- Passing parameters by value
- Variable scope/domain
Functions
What is a function?

- A function is a piece of reusable code.
- You write a function once and can use it many times.
- For example: you can write a function to check whether a date is valid. Then you can check whether many dates are valid or not without repeating your code.
Functions aren’t new!

• You have already used functions in your code!

• For example, `printf` and `scanf` are functions! Someone else wrote them and we used them.

• Today we will see how to write our own functions which we can re-use
Functions as a black box

• You can think of functions as a black box

• Functions take input and do something in order to produce output.

• Sometimes we know how the function works (because we wrote it!)

• Sometimes we only know what the function does (for example, printf)
Here we define a function named `square` which returns a number to the 2\textsuperscript{nd} power.

Notice that we define the function \textbf{before} `main`!

Inside `main` we call our function twice and save the results in `y1`, `y2`.

```c
#include <stdio.h>

int square(int x)
{
    return x*x;
}

int main()
{
    int y1 = square(5);
    int y2 = square(6);
    printf("%d %d\n", y1, y2);
    return 0;
}
```
A closer look

```
int square(int x)
{
    return x*x;
}
```

This function’s output is an int

This function is named square

This function takes as input an int named x

The return instruction specifies what the function’s output is

This function’s output is an int

This function is named square

This function takes as input an int named x

The return instruction specifies what the function’s output is
A more complicated example

The type returned by the function appears before the name

The first line is called the function \textit{declaration} or \textit{signature}. It contains everything you must know to \textbf{use} the function.

\begin{verbatim}
float delta(float a, float b)
{
    float result = a - b;
    if (result < 0)
        result = -result;
    return result;
}
\end{verbatim}

Function name

In parentheses: list of input \textbf{parameters} separated by commas.
A more complicated example

Now let's look at the function **body** or **definition**

This contains code for **how** the function works

```c
float delta(float a, float b) {
    float result = a - b;
    if (result < 0)
        result = -result;
    return result;
}
```

Local variable: Exists only inside the function, while the function is running. It is "forgotten" after the function finishes.

The **return** instruction provides the returned **value**

Function body
The full example

• First we define the function

• Then we call the function inside main()

• We actually have two functions defined here!

• main() is a function too!

```c
#include <stdio.h>

float delta(float a, float b)
{
    float result = a - b;
    if (result < 0)
        result = -result;
    return result;
}

int main()
{
    float x = delta(10.0, 15.0);
    printf("%f", x);
    return 0;
}
```
Functions that don’t return anything

• Here we define a function which takes one input (int x), **does something**, but does **not** return a value

• To show that this function has no output we write **void** for the return type

• We can still use **return** but it is optional

```c
#include <stdio.h>

void print_hi(int x)
{
    printf("HI!\n");
    printf("Your number is %d", x);
    // this return is optional!
    return;
}

int main()
{
    print_hi(5);
    return 0;
}
```
Functions that don’t take any inputs

• This function has no inputs or **parameters**

• To show that it takes no parameters we write the signature as:

\[
\text{int print\_hi\_return10()}
\]

• We can also write “void” for the parameters:

\[
\text{int print\_hi\_return10(void)}
\]

```c
#include <stdio.h>

int print_hi_return10()
{
    printf("HI!\n");
    return 10;
}

int main()
{
    int x = print_hi_return10();
    // x is now 10
    return 0;
}
```
Functions with no inputs or outputs

- This function has no inputs (parameters) and also no outputs (return value)

```c
#include <stdio.h>

void print_hi_return10() {
    printf("HI!\n");
}

int main() {
    print_hi_return10();
    return 0;
}
```
Variable names and functions

• Is this allowed?

• We have two variables named “result”!

• This is OK because variables inside the function are different then the variables outside the function!

• There is no connection between the two variables!

```c
#include <stdio.h>

float delta(float a, float b) {
    float result = a - b;
    if (result < 0) {
        result = -result;
    }
    return result;
}

int main() {
    float result = delta(10.0, 15.0);
    printf("%f", result);
    return 0;
}
```
More details about return

- return exits the current function and continues running the code that **called the function**

- The value passed to `return` is the value to be returned to the caller.

- A **copy** of the value is always returned, even if we write the name of a variable.
  e.g, we have a variable `x` with a current value of 3. Executing “return x;” will return the value 3.

- If return is called from within a loop, if, or switch/case, then the current structure **and the function** stop running.
Declaring a function

- If the function is not defined or declared before it is called, then the compiler will have errors.
- Solution 1: Define the function above the first function that calls it.
- Solution 2: Declare the function above the first function that calls it. Define it later in the program.
- A declaration works as follows:

```c
float delta(float a, float b);
```

No need to write the names of the variables:

```c
float delta(float, float);
```

‘;’ is required
Declaring a function - example

```c
#include <stdio.h>

// here we declare the function without defining it
// notice the semicolon ; at the end of the line
int square(int x);

int main()
{
    printf("%d\n", square(5));
    return 0;
}

// here we define the function
int square(int x)
{
    return x*x;
}
```
Parameters and return value

What are the **parameters** and **return values** for each?

```c
int gcd(int a, int b)
float sin(float x)
void print_account(int id, float cash)
float get_time(void);
void print_table();
int get_temperature(today);
```

You can also write

```c
float get_time();
```
What is wrong with the functions?

float min(int a, int b)
    if (a > b)
        return b;
    else
        return a;

Should return a float not an int!

void print_value(int m)
{
    printf("Value=%d\n", m);
    return m;
}

Returns a value but it is supposed to return nothing!

int find_divisor(int num)
{
    int j = num / 2;
    for (; j>1; j--)
    {
        if (!((num % j))
            return 0;
    }
}

In some cases no value is returned!
#include <stdio.h>

float delta(float a, float b) {
    float result = a - b;
    if (result < 0) {
        result = -result;
    }
    return result;
}

int main() {
    float x, y;
    scanf("%f", &x);
    scanf("%f", &y);
    float d = delta(x, y);
    printf("The difference is %f\n", d);
    return 0;
}

1. Start with main function
2. Call scanf function to get a value for x.
3. Perform the above for y.
4. Call the delta function.
5. The parameters get the values that have been passed from the caller.
6. The function returns the value of “result”
7. The returned value is stored in “d”.
8. Perform the function “printf”
9. The program finishes running when “main” finishes running.
Proper function calls

• Which function calls are incorrect?

```c
int gcd(int n, int m);
void print_value(int num);
int dist(float, float);
char get_letter(void);
```

```c
j = gcd(j, j);
result = print_value(i+1);
dist(2.2, 1.5);
printf("input: %c\n", get_letter(k));
get_letter;
```

-- The function declarations

The parentheses tell the compiler that a function is being called. Otherwise it’s a variable.
Calculation of parameters in a call to a function

- What happened when we called \texttt{delta(x,y)}?

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>y</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2.1</td>
<td>1.5</td>
<td>0.6</td>
</tr>
</tbody>
</table>

In C, you pass values to the functions, And not the variables themselves

- So as long as \( x \) is 2.1 and \( y \) is 1.5, \( \text{delta}(x,y) \) is equivalent to \( \text{delta}(2.1,1.5) \).
Calculating parameters

• First, calculate all parameters passed to the function
• For example, in order to calculate the following function:

\[ \text{delta}(\tan(\cos(0.2)), \sin(1.3)); \]

...all of the parameter values are calculated first

• First, calculate \( \tan(\cos(0.2)) \).
  – To do so, first calculate \( \cos(0.2) \). The result: \( 0.98 \).
  – Then, calculate \( \tan(0.98) \). The result: \( 1.49 \).

• The, \( \sin(1.3) \) is calculated. The result: \( 0.96 \).
• Finally we can calculate \( \text{delta}(1.49, 0.96) \). The result: \( 0.53 \).
Example: Symmetric number
Example: Reversing a number

- Definition by examples:
  - 123 reversed is 321
  - 4343 reversed is 3434
  - 8 reversed is 8
  - 80 (and any number divisible by 10) reversed is -1
- Write a function “int reversed(int x)” which receives a positive integer and returns it reversed.
Symmetric number - solution

```c
int reverse(int x)
{
    if (x\%10 == 0)
        return -1;
    int reversed=0;
    while (x>0)
    {
        reversed *= 10;
        reversed += x\%10;
        x /= 10;
    }
    return reversed;
}
```
Example: Symmetric number

- Define a number as being symmetric if:
  - The number has only one digit
  - Or each digit has a matching digit that is equally distant from the center.

- Example:
  - 3 is a symmetric number since it is only one digit
  - 12321 is a symmetric number, because all numbers have a matching number on the other side of the center digit.
  - 1771 is also a symmetric number

- Write a function int symmetric(int x) which receives a positive integer and returns 1 if it is symmetric, and 0 otherwise.
int symmetric(int x)
{
    /* assuming that x is not negative */
    int reverse=0, tmp=x;
    if (x < 10) { /* a single digit */
        return 1;
    }
    while (tmp > 0) {
        reverse = reverse*10 + tmp%10;
        tmp /= 10;
    }
    return (x==reverse);
}
int symmetric(int x) {
    int reversed = reverse(x);
    if (reversed == -1) {
        return 0;
    }
    return x == reversed;
}
Call stack
Know where you came from and where you are going

• Programs start in the main function
• Main calls another function...let’s call it f.
• f calls another function, let’s call it g.
• Etc..

• How does the program know where to return when all of the functions are finished running?
• For example, when g is running, how does the program know to return to f, and not to main?
Know where you came from and where you are going

• For example, here is a code snippet:

```c
int delta_age;
int delta_height;

delta_age = delta(age1, age2);
delta_height = delta(height1, height2);
```

• Here, the delta function is called twice
• And each time the function is called, the program will continue running from a different location.
• How does this happen?
The Stack

• While a program runs, the system manages a stack of calls
• The stack is managed Last-In-First-Out form.
• The following data is maintained on the stack:
  Where to return after a function finishes
  The values of the parameters that were passed into the function
  Local variables of the function
Maximum of two numbers

• The following function will return the maximum between the two values passed to it.

```c
float max2(float a, float b) {
    if (a > b) {
        return a;
    }
    return b;
}
```

• How can we write a function that determines the maximum of 4 numbers?
Maximum of 4 numbers

- We can create max4 using max2.

```c
float max4(float a, float b, float c, float d)
{
    float temp1 = max2(a, b);
    float temp2 = max2(c, d);
    float max = max2(temp1, temp2);
    return max;
}
```

- Notice: In the body of the function there is not a single comparison between numbers!
Maximum of 8 numbers

• How will we implement the maximum of 8 numbers!

```c
float max8(float a, float b, float c, float d,
           float e, float f, float g, float h)
{
    float temp1 = max4(a, b, c, d);
    float temp2 = max4(e, f, g, h);
    float max = max2(temp1, temp2);
    return max;
}
```

• Notice – we used max4 and max2 in order to solve the problem
The stack during runtime

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Starting main</td>
</tr>
<tr>
<td>2</td>
<td>Calling max8</td>
</tr>
<tr>
<td>3</td>
<td>max8 compares the first 4 parameters with a call to max4</td>
</tr>
<tr>
<td>4</td>
<td>max8 compares the first 2 parameters with a call to max2</td>
</tr>
<tr>
<td>5</td>
<td>max4 gets the result of the call to the first max2</td>
</tr>
<tr>
<td>6</td>
<td>max4 compares the second 2 parameters with a call to max2</td>
</tr>
</tbody>
</table>

The image shows a flowchart with steps detailing the execution of a function, `max8(1,8,4,5,9,6,1,3)`, and how the stack evolves during runtime. Each step represents a function call and the parameters passed to those calls.
### The stack during runtime (cont’d)

<table>
<thead>
<tr>
<th>Function</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>max4</code></td>
<td>1,8,4,5</td>
<td>max4 compares the result of the call to the second <code>max2</code></td>
</tr>
<tr>
<td><code>max8</code></td>
<td>1,8,4,5, 9,6,1,3</td>
<td>max4 gets the result of the call to the second <code>max2</code></td>
</tr>
<tr>
<td><code>main</code></td>
<td>none</td>
<td></td>
</tr>
<tr>
<td><code>max4</code></td>
<td>1,8,4,5</td>
<td>max4 compares between the two results using a call to <code>max2</code></td>
</tr>
<tr>
<td><code>max8</code></td>
<td>1,8,4,5, 9,6,1,3</td>
<td>max4 finishes computing max of the first 4 numbers</td>
</tr>
<tr>
<td><code>main</code></td>
<td>none</td>
<td></td>
</tr>
<tr>
<td><code>max4</code></td>
<td>1,8,4,5</td>
<td>max8 gets the result of the comparison of the first 4 numbers</td>
</tr>
<tr>
<td><code>max8</code></td>
<td>1,8,4,5, 9,6,1,3</td>
<td>max4 compares the next 4 parameters with a call to <code>max4</code></td>
</tr>
<tr>
<td><code>main</code></td>
<td>none</td>
<td></td>
</tr>
<tr>
<td><code>max4</code></td>
<td>9,6,1,3</td>
<td>max8 compares between first 2 parameters with a call to <code>max2</code></td>
</tr>
<tr>
<td><code>max2</code></td>
<td>9,6</td>
<td>max2 compares between the two results using a call to <code>max2</code></td>
</tr>
<tr>
<td><code>max4</code></td>
<td>9,6,1,3</td>
<td></td>
</tr>
<tr>
<td><code>max8</code></td>
<td>1,8,4,5, 9,6,1,3</td>
<td></td>
</tr>
<tr>
<td><code>main</code></td>
<td>none</td>
<td></td>
</tr>
<tr>
<td><code>max4</code></td>
<td>9,6,1,3</td>
<td></td>
</tr>
<tr>
<td><code>max8</code></td>
<td>1,8,4,5, 9,6,1,3</td>
<td></td>
</tr>
<tr>
<td><code>main</code></td>
<td>none</td>
<td></td>
</tr>
</tbody>
</table>
The stack during runtime (cont’d)

- **max8** compares the next 4 parameters using **max4**
- **max4** compares between the next 2 parameters with a call to **max2**
- **max2** params: 1,3
- **max4** params: 9,6,1,3
- **max8** params: 1,8,4,5, 9,6,1,3
- **main** params: none

- **main** params: none
- **max2** gets the result of the first **max2**
- **max4** params: 9,6,1,3
- **max8** params: 1,8,4,5, 9,6,1,3
- **main** params: none

- **main** params: none
- **max4** gets the result of the second call to **max2**
- **max2** params: 9,3
- **max4** params: 9,6,1,3
- **max8** params: 1,8,4,5, 9,6,1,3
- **main** params: none

- **main** params: none
- **max8** compares between the two results with **max2**
- **max8** params: 1,8,4,5, 9,6,1,3
- **main** params: none
- **main** params: none
max8 compares between the two results with a call to max2.

max8 calls max2 directly here. Using the stack, max2 always knows where to return to. Even though it is sometimes being called from max8 and sometimes being called from max4.

max8 params: 1,8,4,5, 9,6,1,3

Main params: 1,8,4,5, 9,6,1,3

Return to main

Main params: none

Main params: none

Max8 returns the result to main

Max8 returns the result to main

Intro to C - Tutorial 8
Passing parameters by value (call by value)
Formal Parameters

• The input for each function is 0 or more formal parameters.

• Within the function, the formal parameters are variables.

• Like all variables within a function:
  • The formal parameters are re-initialized with each call to the function.

• The formal parameters only exist within the function.
Actual parameters

• The values passed by the caller are called **actual parameters**.
• Calling a function does not change the caller’s variables.
• In general, the function will not change the caller’s variables.
Actual and formal parameters

- When calling a function, the **actual parameters** are calculated.

Then the **formal parameters** are initialized with the calculated values and the function runs.
call by value

• Given that the actual parameters in the functions are values, we call these types of function calls:

  call by value

• Later we’ll talk about other types of function calls
Actual and formal parameters

• What are the variable values after calling max?

Function: max

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

Calling max from main

```c
int x = 7, y = 8, d;
d = max(x, y);
```

```c
int a = 7, y = 8, d;
d = max(a, y);
```

```c
int a = 7, b = 8, d;
d = max(a, b);
```

```c
int a = 7, b = 8, res;
res = max(a, b);
```
Variable scope
Variable definition inside a block

• What is a code block?
  
  – Every series of instructions inside braces { } (e.g., the body of an if, switch, or loop with more than 1 line).
  
  – A function body (which is also inside braces { })

• A variable is accessible within a block, from its definition, to the end of the block.
  
  – The variable is **not** accessible outside of the block.
Why “local variables”?

• A variable that is declared inside a block (including at the beginning of a function) is called a local variable.

• They are local to the block that declared them! A local variable does not exist outside of the block that declared it!

• Thus, you cannot use it outside the block.
The variable is not accessible outside of the block

• Riddle: What will the program print?

```c
#include <stdio.h>

int main(void)
{
    int a = 10;
    if (a > 0) {
        int temp = 20;
    }
    printf("%d", temp);
    return 0;
}
```
Every time a block is run, the variable is defined from scratch

• What will the next program print?

```c
#include <stdio.h>

int main(void)
{
    int a;
    int i;
    for (i = 0; i < 10; i++) {
        int temp = 0;
        temp++;
        temp++;
        a = temp;
    }
    printf("%d", a);
    return 0;
}
```
The compiler always uses the variable in the innermost block

• What will the program print?

```c
#include <stdio.h>

int main(void)
{
    int a = 10;
    if (a > 0) {
        int a = 0;
        printf("%d\n", a);
    }
    printf("%d", a);
    return 0;
}
```
In an inner block, it is possible to define a variable with the same name as a former variable from an outer block.

- The inner variable **shadows** the outer variable.
- From the inner definition until the inner block ends, any reference to the variable’s name will refer to the inner variable.
- The outer variable cannot be accessed and doesn’t change until the inner block ends.
- After the inner block ends, the variable from the outer block is back in the game with its former value.

This feature is called **shadowing**.

Avoid shadowing because it is confusing.
Shadowing

- In the case of variables with the same name (one “outside” variable and one “inside” variable) the two variables can have different types.

- This works but don’t do it!

```c
#include <stdio.h>

int main(void)
{
    int a = 10;
    if (a > 0) {
        float a = 0.0;
        printf(“%f\n”, a);
    }
    printf(“%d”, a);
    return 0;
}
```
Shadowing in the same block is illegal

- It’s not possible to shadow a variable by declaring another variable of the same name in the same block.

```c
#include <stdio.h>

int main(void)
{
    int a = 10;
    int b = 5;
    int c = 13;
    int a = 12;  // Hide the first definition of a
    printf("%d", a);
    return 0;
}
```
Global Variables

- You can, in C, define variables that are not in any block - outside of any function.
- They are called **global variables**.
- Global variables are accessible from anywhere in the program.
- **Local variables** are accessible only within the block.
- Local variables are much more common so usually “variable” means “local variable”.
- All the variables we have seen so far in this course have been local variables.
Global variables vs. Local variables

- A global variable exists from the start of the program to the end of the program.
- The value that is written to it remains with it until the end of the program, even if we exit the function that wrote the value.
- A local variable of the same name can “hide” a global variable.
- In most cases it’s better to use local variables.
- In general, using a global variable without a good reason is considered bad programming.
Example: Using a global variable

```c
#include <stdio.h>

int lights_on = 0; /* Global variable */

void flip_lights(void) {
    lights_on = !lights_on;
}

void print_light_status(void) {
    if (lights_on)
        printf("Lights are on.");
    else
        printf ("Lights are off.");
}

int main(void) {
    int lights_on = 1; /* Local variable */
    flip_lights();
    print_light_status();
    return 0;
}
```

What will the program print?