Introduction to C

Tutorial 8: Functions
Last week

- Arrays
- Merging sorted arrays
- 2D arrays
Agenda

- Functions
- Call stack
- Passing parameters by value
- Variable scope/domain
Functions
Defining a function

• Function structure

The type returned by the function appears before the name.

Function name:

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

In parentheses: list of params separated by commas.

Function body:

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.
RETURN instruction

- The **return** operation finishes the execution of the current function.
- Continue running from the place that the function was called.
- The value passed to the 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.
- When the function does not return a value ("void" return type) the return does not get a value (and doesn’t even have to be written).

If return is called from within a **loop** or **switch** instead of break, then the **function** stops running (and not just the current structure!).
Declaration of 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 at the end of the line
Calling a function

• To call a function, you must provide a value for each parameter according to their order.
• You may, though it’s not required, use the value returned by the function.
• Any function can call any other function.
• Example:

   \[
   \text{result} = \text{delta}(c/2.0, d) * 50;
   \]
Calculating parameters in a function call

• 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.
What are the parameters and return values for each?

- `int gcd(int n, int m);`
- `float sin(float number);`
- `void print_account(int id, float cash);`
- `float get_time(void);`
- `void print_table();`
- `int get_temperature(today);`

You can also write `float get_time();`
What is wrong with the functions?

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

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

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

The parentheses tell the compiler that a function is being called. Otherwise it’s a variable.
Running the program

```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, y;
    scanf("%f", &x);
    scanf("%f", &y);
    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.
Calculation of parameters in a call to a function

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

\[
\begin{array}{ccc}
\text{x} & \text{y} & \text{d} \\
2.1 & 1.5 & 0.6 \\
\end{array}
\]

\( \text{d} = \text{delta}(2.1, 1.5); \)

In C, you pass values to the functions, And not the variables themselves
Example: Symmetric number
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 equidistant 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 matching number

• Write a function int symmetric(int x) that receives a positive integer and returns 1 if it is symmetric, and 0 otherwise.
```c
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);
}
```
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

- How does the stack look when `max8(1,8,4,5,9,6,1,3)` is being executed from `main`?

```
Starting main

<table>
<thead>
<tr>
<th>Function</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
<td>none</td>
</tr>
<tr>
<td>max8</td>
<td>1,8,4,5,9,6,1,3</td>
</tr>
<tr>
<td>max8</td>
<td>1,8,4,5,9,6,1,3</td>
</tr>
<tr>
<td>max4</td>
<td>1,8,4,5</td>
</tr>
<tr>
<td>max4</td>
<td>1,8,4,5</td>
</tr>
<tr>
<td>max2</td>
<td>4,5</td>
</tr>
<tr>
<td>max4</td>
<td>1,8,4,5</td>
</tr>
<tr>
<td>max4</td>
<td>1,8,4,5</td>
</tr>
<tr>
<td>max4</td>
<td>1,8,4,5</td>
</tr>
<tr>
<td>main</td>
<td>none</td>
</tr>
</tbody>
</table>

Calling max8

max8 compares the first 2 parameters with a call to max2.
max4 compares the first 4 parameters with a call to max4.
max4 compares the second 2 parameters with a call to max2.
max4 gets the result of the call to the first max2.
max4 compares the second 2 parameters with a call to max2.
max4 compares the first 2 parameters with a call to max2.
max4 compares the first 2 parameters with a call to max2.
max4 compares the first 2 parameters with a call to max2.
max4 compares the first 2 parameters with a call to max2.
```
The stack during runtime (cont’d)

<table>
<thead>
<tr>
<th>Function</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>max8</td>
<td>1,8,4,5,9,6,1,3</td>
</tr>
<tr>
<td>max4</td>
<td>1,8,4,5,9,6,1,3</td>
</tr>
<tr>
<td>main</td>
<td>none</td>
</tr>
<tr>
<td>max8</td>
<td>1,8,4,5,9,6,1,3</td>
</tr>
<tr>
<td>max4</td>
<td>1,8,4,5,9,6,1,3</td>
</tr>
<tr>
<td>main</td>
<td>none</td>
</tr>
<tr>
<td>max8</td>
<td>1,8,4,5,9,6,1,3</td>
</tr>
<tr>
<td>max4</td>
<td>1,8,4,5,9,6,1,3</td>
</tr>
<tr>
<td>main</td>
<td>none</td>
</tr>
<tr>
<td>max8</td>
<td>1,8,4,5,9,6,1,3</td>
</tr>
<tr>
<td>max4</td>
<td>1,8,4,5,9,6,1,3</td>
</tr>
<tr>
<td>main</td>
<td>none</td>
</tr>
</tbody>
</table>

max4 compares between the two results using a call to max2

max4 finishes computing max of the first 4 numbers

max8 compares the next 4 parameters with a call to max4

max4 compares between first 2 parameters with a call to max2
The stack during runtime (cont’d)

<table>
<thead>
<tr>
<th>Function</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
<td>none</td>
</tr>
<tr>
<td>max8</td>
<td>1,8,4,5, 9,6,1,3</td>
</tr>
<tr>
<td>max4</td>
<td>9,6,1,3</td>
</tr>
<tr>
<td>compares</td>
<td>2</td>
</tr>
<tr>
<td>between</td>
<td>the next 2 parameters with a call to max2</td>
</tr>
</tbody>
</table>

- max8 compares the next 4 parameters using max4
- max4 compares between the next 2 parameters with a call to max2
- max4 gets the result of the first max2
- max8 compares between the two results with max2
- max4 gets the result of the second call to max2
- max8 compares between the two results with max2

- max8 compares the next 4 parameters using max4
- max4 compares between the next 2 parameters with a call to max2
- max4 gets the result of the first max2
- max8 compares between the two results with max2
- max4 gets the result of the second call to max2
- max8 compares between the two results with max2
The stack during runtime (cont’d)

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

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

Max8 returns the result to main.
- max8 params: 1,8,4,5, 9,6,1,3
- main params: none

Return to main.
- main params: none
Passing parameters by value (call by value)
Actual and formal parameters

• The parameters that the function accepts (based on the function definition) are called formal parameters.

From the perspective of the function, the formal parameters are simply **variables**. They are only recognized within the function, and re-initialized with every call to the function.

• The values passed to the parameters (required in each function call) are called **actual parameters**.

The actual parameter is always a **value**! The function **cannot** change the value of the variable!
• 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

- You may declare variables at the beginning of code block.
- What qualifies as a block?
  - A function body
  - Every series of instructions 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.
- In the case of several runs within a block (for example, a loop):
  - Every time the block is run, the variable is “defined” from scratch.
- In the case of duplicate names of variables, the compiler will always refer to the variable that was declared in the innermost block (relative to points of access of the variable).
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 will refer to the variable that was declared 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;
}
```
“The compiler will refer to the variable that was declared in the innermost block”

• The inner variable **hides** the outer variable when they have the same name.

• This feature is called **hiding** or **shadowing**

• Coding that uses this can be very confusing

• Therefore, we recommend that you avoid it
In the case of variables with the same name (one “outside” variable and one “inside” variable) the two variables can have different types.

Example:

```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;
}
```
Hiding in the same block?

• Do you think it’s possible to hide a variable by declaring another variable of the same name in the same block?
• For example, what will this code print?

```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;
}
```
Variables “outside of the block”

• You can, in C, define variables that are not in any block.
• These variables are defined outside of any function.
• They are called **global variables** because they are accessible from anywhere in the program (in contrast to **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**
#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;
}