

C introduction

# Functions

# Contents

Using functions

More on scopes

Recursion

## Remember the main *function*?


```
int main(void) {  
    /* code happens */  
    return 0;  
}
```

# Defining functions

```
return_type identifier(argument_list) {  
    function_body  
    return expression;  
}
```

# Defining functions

data type of the  
returned value or *void*,  
if nothing is returned




```
return_type identifier(argument_list) {  
    function_body  
    return expression;  
}
```

# Defining functions

data type of the  
returned value or *void*,  
if nothing is returned

unique name to refer  
the function, same rules  
as for variable identifiers




```
return_type identifier(argument_list) {  
    function_body  
    return expression;  
}
```

## Defining functions

data type of the  
returned value or *void*,  
if nothing is returned

unique name to refer  
the function, same rules  
as for variable identifiers

argument declarations,  
seperated by commas or  
*void*, if there are none



```
return_type identifier(argument_list) {  
    function_body  
    return expression;  
}
```

## Defining functions

data type of the  
returned value or *void*,  
if nothing is returned

unique name to refer  
the function, same rules  
as for variable identifiers

argument declarations,  
seperated by commas or  
*void*, if there are none

```
return_type identifier(argument_list) {  
    function_body  
    return expression;  
}
```

The diagram illustrates the syntax of a function definition. Three arrows point from the descriptive text above to the corresponding parts of the code: one from 'data type of the returned value or void' to 'return\_type', one from 'unique name to refer the function' to 'identifier', and one from 'argument declarations' to 'argument\_list'. A red arrow points from the text 'just as in main(), all statements are put in here' to the 'function\_body' line in the code block.

just as in *main()*, all statements are  
put in here



## Defining functions

data type of the  
returned value or *void*,  
if nothing is returned

unique name to refer  
the function, same rules  
as for variable identifiers

argument declarations,  
seperated by commas or  
*void*, if there are none

```
return_type identifier(argument_list) {  
    function_body  
    return expression;  
}
```

The diagram illustrates the syntax of a function definition. A box contains the code template. Arrows point from descriptive text blocks to specific parts of the code: 'data type of the returned value or void, if nothing is returned' points to 'return\_type'; 'unique name to refer the function, same rules as for variable identifiers' points to 'identifier'; 'argument declarations, separated by commas or void, if there are none' points to 'argument\_list'; 'just as in main(), all statements are put in here' points to the 'function\_body' block; and 'value this function returns or empty, if the return value is void' points to the 'return expression;' line.

just as in *main()*, all statements are  
put in here

value this function returns or empty,  
if the return value is *void*

# Defining functions

data type of the  
returned value or *void*,  
if nothing is returned

unique name to refer  
the function, same rules  
as for variable identifiers

argument declarations,  
seperated by commas or  
*void*, if there are none

```
return_type identifier(argument_list) {  
    function_body  
    return expression;  
}
```

The diagram illustrates the components of a function definition. Arrows point from the following text blocks to specific parts of the code: 'data type of the returned value or void' points to *return\_type*; 'unique name to refer the function' points to *identifier*; 'argument declarations' points to *argument\_list*; 'just as in main(), all statements are put in here' points to the *function\_body* block; and 'value this function returns or empty' points to the *return expression* statement.

just as in *main()*, all statements are  
put in here

value this function returns or empty,  
if the return value is *void*

## Passing arguments

- ▶ Each value is assigned to the parameter at the same position in the argument list (and therefore must have the same type)

```
1 #include <stdio.h>
2
3 void shift_character(char character, unsigned offset) {
4     printf("%c\n", (character + offset) % 255);
5 }
6
7 int random_number(void) {
8     return 4;    // chosen by fair dice roll.
9                 // guaranteed to be random.
10 }
11
12 int main(void) {
13     int offset = 10;
14     shift_character('c', offset);
15     printf("%d\n", random_number());
16     return 0;
17 }
```

## Global variables

- ▶ Variables defined outside any function
- ▶ Scope: from line of declaration to end of program

```
1 int globe = 42;
2
3 void foo(void) {
4     globe = 23;
5 }
6
7 int main(void) {
8     printf("%d\n", globe); /* Prints 42 */
9     foo();
10    printf("%d\n", globe); /* Prints 23 */
11    ...
```

Altering them in one function may have side effects on other functions → use them rarely.

## Where not to call functions

Since a function's scope starts at the line of its definition, having two functions  $f()$  and  $g()$  calling each other is not possible:

```
1 void f(int i) {  
2     ...  
3     g(42); /* What is g? */  
4 }  
5  
6 void g(int i) {  
7     ...  
8     f(42);  
9 }
```

In that case,  $g()$  is called outside its scope. Changing the order does not work either.

# Prototypes

Like variables, functions can also be *declared*:

```
return_type identifier(argument list);
```

- ▶ It's similar to a definition, just replace the function body by a ;
- ▶ Declared functions must also be defined any where in the program
- ▶ In the argument list, only types matter → identifiers can be left out

```
1 void g(int i); /* better do not leave the identifier out */
2
3 void f(int i) {
4     ...
5     g(42);      /* Now a call of g() can be compiled */
6 }
7
8 void g(int i) {...} /* g() definition , similar to f() */
```

## Better program structure

To avoid problems like that above, it is a common practise to *declare* all functions at the top of the file and define them below the main function:

```
1 void f(int i);  
2 void g(int i);  
3  
4 int main(void) {  
5     ...  
6 }  
7  
8 void f(int i) {  
9     ...  
10    g(42);  
11 }  
12  
13 /* g() definition , similar to f() */
```

## Good documentation style

Add a documentation comment to each function prototype:

```
1 /*  
2  * Get the sum of two numbers.  
3  * num:      input number  
4  */  
5 int factorial(int num);
```

There are frameworks such as *doxygen* that parse your comments and create a fancy HTML documentation:

```
1 /**  
2  * @brief Get the sum of two numbers.  
3  * @param num1    first number  
4  * @param num2    second number  
5  * @return        sum of num1 and num2  
6  */  
7 int add(int num1, int num2);
```



## Functions in functions

You could define functions in functions.<sup>1</sup>

---

<sup>1</sup>Just saying.

# Recursive functions

- ▶ Functions calling themselves
- ▶ Used to implement many mathematical algorithms
- ▶ Easy to think up, but they run slow

Careful:

```
1 void foo(void) {  
2     foo();  
3 }
```

creates an infinite loop.<sup>2</sup>

There must always be an *exit condition* if using recursion!

---

<sup>2</sup>And, at some point, a program crash (*stack overflow*)

# Exponentiation

As an example, take a look at this function calculating  $base^{exponent}$ :

```
1 int power(int base, int exponent) {  
2     if (exponent == 0)  
3         return 1;  
4     return base * power(base, exponent - 1);  
5 }
```

- ▶  $a^0 = 1 \rightarrow power(a, 0)$  just returns 1
- ▶  $a^b = a * a^{b-1} \rightarrow$  recursive call of  $power(a, b-1)$