

C introduction

Functions

Contents

More rabbits

- ▶ Write a program that prints the 3rd, 5th, 11th and 5th Fibonacci number.

More rabbits

- ▶ Write a program that prints the 3rd, 5th, 11th and 5th Fibonacci number.
- ▶ It looks like a multiple repetition of

```
1  for (i = 0, fib1 = 0, fib2 = 1; i < n; i++) {  
2      int buffer = fib1;  
3      fib1 = fib2;  
4      fib2 = fib2 + buffer;  
5  }  
6  printf(" fib(%d) = %d\n", i, fib1);
```

There must be a better way

Everytime you type an exact copy of a line of code, you should *take a break* and rethink what you're doing.

Functions are a powerful tool to structure your code and avoid repetitions.

Functions serve the *divide and conquer* programming paradigm - breaking down a problem into sub-problems easier to solve.

Theory crafting

Since we need *fib(5)* twice, we can imagine a function:

```
Function fib_5:  
    // calculate fib(5) here and print it
```

Theory crafting

Since we need *fib(5)* twice, we can imagine a function:

```
Function fib_5:  
    // calculate fib(5) here and print it
```

As we want to separate calculation from output, consider the function *returning* its result to the main function, where it is printed:

```
Function fib_5:  
    // calculate fib(5) here  
    return result;
```

Theory crafting

Since we need *fib(5)* twice, we can imagine a function:

```
Function fib_5:  
    // calculate fib(5) here and print it
```

As we want to separate calculation from output, consider the function *returning* its result to the main function, where it is printed:

```
Function fib_5:  
    // calculate fib(5) here  
    return result;
```

Since the calculation of certain Fibonacci numbers only differs in the *parameter* *n*, we might want to pass it to our function:


```
Function fib(int n):  
    // calculate fib(n) here  
    return result;
```


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

parameter declarations,
seperated by commas
(e.g. *int a, char b*)



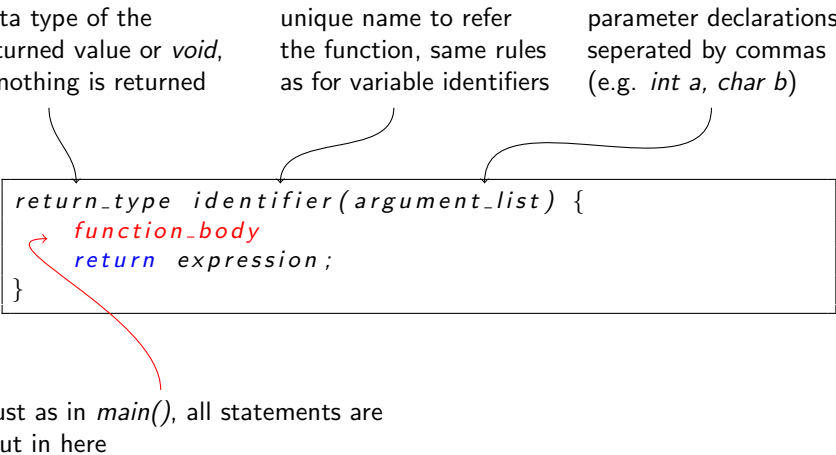
```
return_type identifier(argument_list) {  
    function_body  
    return expression;  
}
```

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

parameter declarations,
seperated by commas
(e.g. *int a, char b*)



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

The diagram illustrates the syntax of a function definition. It consists of a code block with three lines: a return type, a function name with arguments in parentheses, an opening curly brace, a function body, a return statement, and a closing curly brace. Arrows point from descriptive text to these components: 'return_type' points to the first line, 'identifier' points to the function name, 'argument_list' points to the parentheses, 'function_body' points to the second line, 'return expression;' points to the third line, and '}' points to the closing brace. The words 'function_body' and 'return' are highlighted in red in the original image.

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

parameter declarations, separated by commas (e.g. *int a, char b*)

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

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

parameter declarations,
separated by commas
(e.g. *int a, char b*)

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

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

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

Example (1)

The function that calculates *fib(5)* and prints it directly:

```
1 void fib_5 () {  
2     int i;  
3     int fib1;  
4     int fib2;  
5     for (i = 0, fib1 = 0, fib2 = 1; i < 5; i++) {  
6         int buffer = fib1;  
7         fib1 = fib2;  
8         fib2 = fib2 + buffer;  
9     }  
10    printf(" fib(%d) = %d\n", i, fib1);  
11 }
```

It takes no arguments and returns nothing. Note: the *return* statement at the end can be left out completely in *void* functions.

Example (2)

The function that calculates *fib(n)* and returns the result:

```
1 int fib(int n) {  
2     int i;  
3     int fib1;  
4     int fib2;  
5     for (i = 0, fib1 = 0, fib2 = 1; i < n; i++) {  
6         int buffer = fib1;  
7         fib1 = fib2;  
8         fib2 = fib2 + buffer;  
9     }  
10    return fib1;  
11 }
```

It takes an *int* argument and returns an *int* value.

Call of functions

You can call a function in a statement by typing its name followed by ():

```
fib_5 ();
```

Arguments must be written inside the parentheses, separated by commas:

```
fib (5);
```

The return value can be used as a part of a statement or assigned to a variable of the same type:

```
printf("fib(%d) = %d\n", 5, fib(5));  
int result = fib(5);
```

Passing arguments

You must pass as many arguments as declared in the function definition. Each value is assigned to the parameter at the same position in the argument list (and therefore must have the same type):

```
1 void foo(int value, char dec1, char dec2) {  
2     /* things happen */  
3 }  
4  
5 ...  
6  
7 int main(void) {  
8     int number = 42;  
9     char dec1 = '4';  
10    char dec2 = '2';  
11    foo(number, dec1, dec2);  
12    return 0;  
13 }
```

Now it's your turn

Take a look back at our Fibonacci numbers program and apply all the changes we discussed!

Now it's your turn

Take a look back at our Fibonacci numbers program and apply all the changes we discussed!

```
1 int main(void) {  
2     printf(" fib(%d) = %d\n", 3, fib(3));  
3     printf(" fib(%d) = %d\n", 5, fib(5));  
4     printf(" fib(%d) = %d\n", 11, fib(11));  
5     printf(" fib(%d) = %d\n", 3, fib(3));  
6     return 0;  
7 }
```

Feels a lot cleaner, doesn't it?

Global variables

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

```
1 int globe = 42;
2
3 void foo() {
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() */
```

Functions in functions

You **could** define functions in functions.¹

¹Just saying.

Recursive functions

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

Careful:

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

creates an infinite loop.²

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

²And, 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 $power(a, b-1)$

Exercises

- ▶ You are now able to solve tasks 10 and 11.