CS Basics
9) Functions and Arrays

Fall Term 2023-24

E.Benoist, C.Fuhrer, Ch. Grothoff, L. Ith, P.Mainini | BFH-TI

Functions and Arrays

- **Functions**
  - Function Definition
  - Function Prototypes
  - Static Functions
  - Libraries
- **Variable Scope**
  - Automatic Variables
  - Static Variables
  - Global / External Variables
- **Arrays**
- **Bitwise Operations**
- **Conclusion**

Functions

Functions: Overview

Are used to group instructions

- Break a program down into smaller, logical components
- Functions should be *self-contained* and *not too large*
- Help with modularization of programs

Calling functions

- A function can in general be called from anywhere in the program
- This avoids writing the same code multiple times in different places

Makes it easier to manage complex code

- No need to debug redundant code

Allows the programmer to define custom libraries

- Functions can be grouped together to be reused
- Thousands of free software libraries exist already

Function Definition: Ingredients

A function definition consists of two required parts

- *Function header* (return type, function name and parameters)
- *Body* of the function

Syntax of the function header

```
type fname(type1 par1, type2 par2, ..., typeN parN)
```

The body

- Consists of a *compound statement*

Function Definition: Example

`lower2upper()`: **transforms a character from lower case to upper case**

```
char lower2upper(char c) {
  char result = c;
  int offset = 'A' - 'a';

  if ((c >= 'a') && (c <= 'z'))
    result += offset;

  return result;
}
```

Function Definition: Return Type

A function can return a value or nothing When returning a value...

- *Return type* must be given in the function header
- Any type is allowed (except arrays)
- Always explicitly `return` a value to avoid unwanted behavior
- `return` statements must return a value of the defined type

When returning no value...

- The return type *should* be set to `void`!
- Beware: `int` is taken as default when no type is specified!
- No explicit `return` statement is required

Function Definition: No Parameters

A function is not required to have any parameters

- E.g. functions working only with global variables or performing output only…
- Parameter list *should* be set to void!

Syntax

```
type fname(void)
```

Example

```
char read_char(void) {
  char res;
  scanf("%c", &res);
  return res;
}
```

Example: Non-`void` Parameter List

```c
void f1() {}

void f2(void) {}

int main(void) {
  // no compiler error
  f1("foo", "bar");

  // error: too many arguments to function 'f2'
  f2("foo", "bar");
}
```

Function Prototypes (I)

A function must be declared *before* it can be called

- Seen so far: function definition prior to any call
- Otherwise: compiler does not know the *symbol*

In practice, definition before usage is not always possible

- `function_a()` calls `function_b()`
- …which calls `function_c()`
- …which in turn calls `function_a()` again
- → finding total order is not possible in every case

Solution: function prototypes or declarations

- Only declare a symbol without providing the implementation
- Consists only of function header (return type, function name and parameter *types*) *without* body, but with "`;`"
- Parameter *names* may be omitted

Function Prototypes (II)

```
// function prototype
void func_a(char);

// function definitions
void func_b(void) {
  puts("B");
  func_a('A');
}

void func_a(char c) {
  printf("%c\n", c);
  func_b();
}

// main function
int main(void) {
  func_a('A');
}
```

Static Functions

Static functions are limited in scope to the current *compilation unit*

- *Functions should be declared* `static`*, unless they are intended to be used from another compilation unit!*
- Limiting the scope allows reuse of function names
- Proper grouping of functions into compilation units considers reduction of non-`static` functions
- Libraries should not export internal functions
- Not exporting symbols may result in smaller, faster binaries

Example

```
static void helper(void) {
  printf("Hello world!\n");
}

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

Using Libraries

Using library functions requires their prototypes

- They are defined in header files (`.h`), e.g. `stdio.h`, `math.h`
- Headers must be included before using the functions

Example without including `stdio.h`

- Just the prototypes are required…

```
// define protoypes locally, no #include
int printf(const char *, ...);
int scanf(const char *, ...);

int main(void) {
  printf("Hello, what is your name?\n");
  char name[80] = {0};
  if (1 != scanf("%79s", name))
    return 1;

  printf("Hello %s!\n", name);
}
```

The Linking Process

The implementation (body) of a function is normally not in a header file

- A header file typically consists only of function prototypes
- May also contain definitions of global variables, constants and type definitions (introduced later)

Typically, the implementation is compiled only once

- Speeds up compilation
- Stored as object (`.o`), *shared library*/*shared object* (`.so`) or something similar somewhere on the system
- *Linked* by `gcc` during linking stage

Programs Spanning Multiple Files

Different functions may be defined in multiple source files

- Typically grouped by function / field of application
- Functions which operate on the same data structure are often in the same file
- A single source file should not become too large
- *Functions that are used only within the same file should be declared as* `static`

Often, for each `.c` file, a corresponding header file (`.h`) is written

- Contains the *exported* function prototypes, constants and data structures of the `.c` file

Example: `library.c`

```c
uint64_t factorial(uint64_t val) {
  uint64_t result = 1;

  for (uint64_t i = 1; i <= val; i++) {
    result *= i;
  }

  return result;
}

uint64_t logarithm(uint64_t val) {
  if (val <= 1)
    return 0;

  uint64_t result = 0;
  while (val > 1) {
    val /= 2;
    result++;
  }

  return result;
}
```

Example: library.h

```c
// header file for library.c
#ifndef H_LIBRARY
#define H_LIBRARY

#include <stdint.h>

uint64_t factorial(uint64_t);
uint64_t logarithm(uint64_t);

#endif
```

Example: `library_main.c`

Programs using functions from `library`

- Must include the function prototypes from `library.h`
- Can then use exported functions, constants etc.

Example

```c
#include "library.h"
#include <stdio.h>

int main(void) {
  puts("Please enter a number greater than 0:");

  uint64_t num = 0;
  if (1 != scanf("%lu", &num))
    return 1;

  if (num == 0) {
    puts("Logarithm is not defined for 0!");
    return 2;
  }

  printf("You entered: %lu\n", num);
  printf("log_2(%lu) = %lu and %lu! = %lu\n", num, logarithm(num), num,
         factorial(num));
}
```

Compiling and Linking with a Library

All files have to be compiled separately

- Every `.c` file is compiled into a separate `.o` file

Linking

- Every time a library is used, its `.o` file(s) must be linked with the `.o` file containing the `main()` function

Makefile Example

```
library_main: library_main.o library.o
  $(CC) $(CFLAGS) -o library_main library_main.o library.o

library_main.o: library_main.c library.h
  $(CC) $(CFLAGS) -c library_main.c

library.o: library.c
  $(CC) $(CFLAGS) -c library.c
```

Makefile.am (Autotools)

```
bin_PROGRAMS = \
  myprog

myprog_SOURCES = \
  my_program.c
myprog_LDADD = \
  libmy.la

lib_LTLIBRARIES = \
  libmy.la
libmy_la_SOURCES = \
  my_lib.c
libmy_la_LIBADD = \
  -lm
include_HEADERS = \
  my_lib.h \
```

Variable Scope

Automatic Variables

Variables defined *within a function* have *automatic storage duration*

- The same applies to *function parameters*
- Lifetime and accessibility of automatic variables is limited by enclosing scope ({})
- May optionally be declared as `auto` (unnecessary)

Example

```
void func1(void) {
  auto char c = 0, d = 0;
  auto double x = 42.0;
  // ...
}
```

is equivalent to

```
void func1(void) {
  char c = 0, d = 0;
  double x = 42.0;
  // ...
}
```

Example: Function Scope

```c
#define MAX_STARS 10

void print_stars(int nb_stars) {
  int i;
  printf("%d)", i); // warning: 'i' is used uninitialized

  for (i = 0; i < nb_stars; i++) {
    putchar('*');
  }
  putchar('\n');
}

int main(void) {
  for (int i = 0; i < MAX_STARS; i++) {
    print_stars(i);
  }
}
```

Example: Nested Statements

```c
#define ITERATIONS 10

int main(void) {
  for (int i = 0; i < ITERATIONS; i++) {
    if (i < 5) {
      int i = 9; // BAD IDEA
      printf("i=%d, ", i);
    } else {
      printf("i=%d, ", i);
    }
  }
  putchar('\n');
}

// Output:
// i=9, i=9, i=9, i=9, i=9, i=5, i=6, i=7, i=8, i=9,
```

Static Variables

Static variables are allocated for the whole *duration of the program*

They are accessible from the scope of their definition

- This is typically the source file or a function

Opposed to automatic variables, they are *initialized to o by default*

Stored in `.data` or `.bss` segment

If an initializer is provided, it is run only once

Static variables *must not* be declared in headers

Example:

```
static uint32_t number = 42;
```

Example: Static Variables

```c
int counter(void) {
  static int i = 0;
  i++;
  return i;
}

int main(void) {
  counter();
  counter();
  counter();
  counter();
  counter();
  printf("counter = %d\n", counter());
}

// Output:
// counter = 6
```

Global Variables

Global variables are defined outside any function

- Can be accessed from anywhere after their definition
- Are initialized to o by default, like static variables
- Also stored in `.data` or `.bss` segment

Example

```c
int my_var;

static void print_var(void) { printf("my_var = %d\n", my_var); }

int main(void) {
  my_var = 10;
  print_var();

  my_var /= 2;
  print_var();
}

/*
  Output:
  my_var = 10
  my_var = 5
*/
```

External Variables

External variables can be accessed from other files

- Storage-class *external*, defined using "`extern`"
- Used for declaring global variables in header files
- Must be defined *exactly once* without "`extern`" in one of the `.c` files

Example (see next slide for external definitions)

```
extern int my_var1;

void print_var(void) {
  extern int my_var2;

  printf("my_var1 = %d\n", my_var1);
  printf("my_var2 = %d\n", my_var2);
}
```

Example: External Variables (Def.)

```c
void print_var(void);

int my_var1;
int my_var2;

int main(void) {
  my_var1 = 10;
  my_var2 = -1;
  print_var();

  my_var1 /= 2;
  my_var2 *= 2;
  print_var();
}

/*
  Output:
  my_var1 = 10
  my_var2 = -1
  my_var1 = 5
  my_var2 = -2
*/
```

Arrays

Introducing Arrays

Arrays are a data structure for storing elements of the *same* type

- Access time is constant
- The first element has index 0
- Last element has index *(length of array) - 1*
- *The size of an array cannot change after definition!*

An array variable basically represents a start address in memory

- Can be seen as a label for a memory region (similar to assembly)
- a[0] is the first element (at address "`a`")
- a[5] is the *sixth* element
  (at address "`a + 5 * sizeof a`")

Defining Arrays

- To define an array, its type and the number of elements (i.e. the array length) must be specified
- Syntax:

```
type array_name[length];
```

- Examples:

```
// Defining an array containing five double values
double balance[5];

// Defining and initializing an array in one statement
double balance[5] = {1000.0, 2.0, 3.4, 7.0, 50.0};

// When initializer is provided, length may be omitted
double balance[] = {1000.0, 2.0, 3.4, 7.0, 50.0};

// If initializer is too small, remaining values are set to 0
// e.g. {1000.0, 2.0, 0.0, 0.0, 0.0}:
double balance[5] = {1000.0, 2.0}

// Setting the value of an array element
balance[4] = 50.0;
```

Accessing Elements

Array elements can be accessed using the `[]` operator;

accesses memory at "`array_start + [index] * sizeof element`"

Example:

```
#define ARRAY_SIZE 5

int main(void) {
  double array1[ARRAY_SIZE] = {1.3, 4.5, 8.9, -1.0, 3};

  for (int i = 0; i < ARRAY_SIZE; i++) {
    printf("array1[%d]=%lf\n", i, array1[i]);
  }
}
```

Multidimensional Arrays (I)

The C programming language allows for multidimensional arrays

Syntax:

```
type array_name[size1][size2]...[sizeN];
```

Example:

```
int8_t myarray[5][10][4];
```

Multidimensional Arrays (II)

Definition:

```
int8_t a[3][4];
```

Initialization:

```
int8_t a[3][4] = {
  {0, 1, 2, 3},     /*  initializers for row 0 */
  {4, 5, 6, 7},     /*  initializers for row 1 */
  {8, 9, 10, 11}    /*  initializers for row 2 */
};
// in memory: {0,1,2,3,4,5,6,7,8,9,10,11}
```

Accessing elements:

```
int8_t v = a[1][2]; // v=6
```

Example: Iterate 2-dimensional Array

```c
int main(void) {
  // array with 5 rows and 2 columns
  int array2[5][2] = {{0, 0}, {1, 2}, {2, 4}, {3, 6}, {4, 8}};
  // output value of each array element
  for (int r = 0; r < 5; r++)
    for (int c = 0; c < 2; c++)
      printf("array2[%d][%d] = %d\n", r, c, array2[r][c]);
}
/*
  Output:
  array2[0][0] = 0
  array2[0][1] = 0
  array2[1][0] = 1
  array2[1][1] = 2
  array2[2][0] = 2
  array2[2][1] = 4
  array2[3][0] = 3
  array2[3][1] = 6
  array2[4][0] = 4
  array2[4][1] = 8
*/
```

Using Arrays as Function Parameters

Unsized array parameter

```
void func(char param[]) {
  // ...
}
```

Sized array parameter : Required for multidimensional arrays

```
void func(char param[10][10]) {
  // ...
}
```

Pointer to array[1]

```
void func(char *param) {
  // ...
}
```

[1] Pointers will be introduced in lesson C-4!

Example: Array Function Parameter

```c
static double average(double array[], size_t len) {
  if (len == 0)
    return 0.0;

  double sum = 0.0;
  for (size_t i = 0; i < len; i++)
    sum += array[i];

  return sum / len;
}

int main(void) {
  double balance[5] = {1337, 42.20, 23.30, 17, 53.85};
  double avg = average(balance, 5);
  printf("Average is: %.2f\n", avg);
}

// Output:
// Average is: 294.67
```

Returning an Array from a Function

Return value is a *pointer* to an element of the array type (“`type *`”)

- Normally points to the the first element of the array to return

Syntax

```
char *func(...) {
  // ...
}
```

Example: Array as Return Value

```c
int *random_numbers(void) {
  static int r[10];

  // seed (initialize) the random number generator
  srand((unsigned int)time(NULL));

  for (int i = 0; i < 10; i++) {
    r[i] = rand(); // do not use for cryptography!
    printf("r[%d] = %d\n", i, r[i]);
  }

  // only return pointers to static variables!
  return r;
}

int main(void) {
  int *p = random_numbers();
  for (int i = 0; i < 10; i++) {
    printf("*(p + %d) : %d\n", i, *(p + i));
  }
}
```

Variable Length Arrays (VLA, C99)

Normally, the size of an array is defined at *compile time*

C99 introduces *variable length arrays* which may be defined using a variable size[2]

Works only for automatic arrays and array size is still fixed *after* definition

Beware of size and lifetime: **array is allocated on the stack!**

Example:

```
puts("Enter VLA length:"); // not so smart...
unsigned int len = 0;
if (1 == scanf("%u", &len)) {
  int array[len];
  printf("len: %d, size: %ld\n", len, sizeof array);
}
```

[2] And C11 makes them optional again…

Bitwise Operations

Bitwise Operations

Operate on individual bits

Equivalent to the corresponding assembly instructions

Logical operations:

- `~` : Negation (“NOT”, one’s complement)
- `&` : AND
- `|` : OR
- `^` : XOR

Negation

Negate a number bitwise

- A number is interpreted as array of bits
- 0xF5 = 11110101
- ~0xF5 = 00001010 = 0x0A

Example

```
uint16_t n1 = 0x1;
uint16_t n2 = ~n1; // one's complement
printf("NOT %04X = %05X\n", n1, n2);

n1 = 0xF0F0;
n2 = ~n1;
printf("NOT %04X = %05X\n", n1, n2);

/*
  Output:
  NOT 0001 = 0FFFE
  NOT F0F0 = 00F0F
*/
```

Example: Binary, Bitwise Operations

```
  uint16_t n1 = 0xFF11;
  uint16_t n2 = 0x5599;

  uint16_t n3 = n1 & n2;
  printf("%X AND %X = %05X \n", n1, n2, n3);

  uint16_t n4 = n1 | n2;
  printf("%X OR %X = %05X \n", n1, n2, n4);

  uint16_t n5 = n1 ^ n2;
  printf("%X XOR %X = %05X \n", n1, n2, n5);

/*
  Output:
  FF11 AND 5599 = 05511
  FF11 OR 5599 = 0FF99
  FF11 XOR 5599 = 0AA88
*/
```

Bit Shifts

Shift the bits of a number

- A number is interpreted as array of bits
- Bits are shifted either left or right

The << operator

- `x << n`: shifts `x n` bits to the left

The >> operator

- `x >> n`: shifts `x n` bits to the right

Example: Bit Shifts

```c
  uint32_t a = 0xFF01;

  printf("uint32_t a = %08X\n\n", a);
  printf("a << 3     = %08X\n", a << 3);
  printf("a >> 3     = %08X\n", a >> 3);
  printf("a << 4     = %08X\n", a << 4);
  printf("a >> 4     = %08X\n", a >> 4);
  printf("a << 31    = %08X\n", a << 31);

  // warning: left shift count >= width of type [-Wshift-count-overflow]
  printf("a << 32    = %08X (undefined!)\n", a << 32);


/*
  Output:
  uint32_t a = 0000FF01

  a << 3     = 0007F808
  a >> 3     = 00001FE0
  a << 4     = 000FF010
  a >> 4     = 00000FF0
  a << 31    = 80000000
  a << 32    = 0000FF01 (undefined!)
*/
```

Conclusion

Conclusion

Programs can (and should) be split across multiple files

- Files should contain functions that belong together
- Libraries: even larger collections of functions
- `mylib.h`: declarations of the functions
- `mylib.c`: definition of the functions
- `mylib.h`: must be included in all files where library functions are used
- `mylib.o`: must be linked where library functions are used

Variables

- Automatic: defined in a block (and blocks inside)
- Global / external: visible everywhere (even in other files)
- Static: initialized only once for the lifetime of a program (even if the function is called many times)

Conclusion

Arrays

- Structure for storing in memory a collection of elements of the same type.

Bitwise Operations

- Possibility to work at the bit level
- Shift, logical bitwise operators (AND, OR, XOR, NOT)