Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

CS Basics
11) Functions Arrays

E. Benoist & C. Grothoff
Fall Term 2018-19

Computer Science Division

Functions and Arrays

- Functions
  - Functions Definitions
  - Function Prototype
  - Libraries
- Scope of variables
  - Automatic Variables
  - Static variables
  - Global / Extern
- Arrays
- Low Level Programming
- Conclusion

Functions

Functions

- **They are defined to group instructions**
  - A program must be broken into number of smaller components
  - Each component (i.e. function) must be “self-contained”
  - This is like a “modularization” of the program
- **Functions call**
  - The same function can be called from anywhere in the program
  - It avoids having to write many times the same code.
- **Code easier to manage**
  - No need to debug redundant code
- **Define your own customized libraries**
  - Group your functions together to reuse them
  - Thousands of free software libraries exist already

Functions Definitions

Example of function definition

- **lower2upper() to transform a lower case character into a upper case character**

```
#include <stdio.h>

char lower2upper(char c){
  int delta = 'A'-'a';
  char res=c;
  if((c>='a') && (c<='z')){
    res += delta;
  }
  return res;
}
int main(){
  char c1;
  while(scanf("%c",&c1)!=EOF){
    c1 = lower2upper(c1);
    printf("%c",c1);
  }
}
```


Defining a function

- **Two principal components**
  - Function header (return type, name and arguments)
  - The body
- **Function header**
  - `type myName(type1 arg1,type2 arg2,..., typen argn)`
  - If types are omitted, int is a default type (very bad practice)
- **Body**
  - Compound statement
- **Example**

```
char lower2upper(char c){
  int delta = 'A'-'a';
  char res=c;
  if((c>='a') && (c<='z')){
    res += delta;
  }
  return res;
}
```

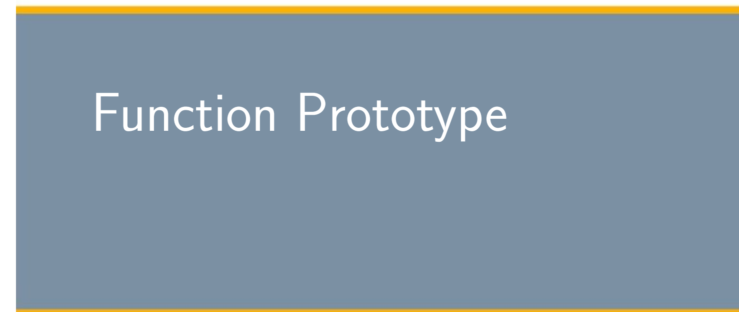

Return type

- **A function can return a value**
  - Return type must be written before function name
  - Can be any type
  - Each `return` statement must return an element of the given type.
- **Or return no value**
  - The return type is then `void`
  - A function body without `return` is then also valid

Functions with no Arguments

- **A function may have no arguments**
  - Subroutines
  - Work only with global variables
  - Input / Output only
  - ...
- **Syntax**
  - `dataType functionName(void)`
- **Example**

```
long readLong(void){
  long res;
  scanf("%ld",&res);
  return res;
}
```

Function Prototype

- **A function must be known before it is used**
  - We have defined our functions prior to calling them
  - Otherwise: compiler does not know the *symbol*
- **Problem**
  - Not so easy to determine the order to define the functions
  - functionA calls functionB,
  - which calls functionC
  - functionC calls functionA
  - Impossible to find total order!
- **Solution: the prototype**
  - Defines the symbol without providing an implementation
  - Declares return type, function name and arguments. Body is replaced with semicolon (i.e. ";")
  - Must be used for mutually recursive functions

Function Prototype (Cont.)

```
/* Function Prototypes */
int funcA(void);
int funcB(void);
int funcC(void);

/* Function Definitions */
int funcA(void){
  puts("A");
}
int funcB(void){
  puts("B");
  funcA();
}
int funcC(void){
  puts("C");
  funcB();
}
/* Main function */
int main(){
  funcC();
}
```

Libraries

Use of Libraries

- **Library functions prototypes**
  - Inside the `.h` file : `stdio.h`, `math.h`, ...
  - Must be inserted on top of the files where they are used
  - Is equivalent to the following:
- **Example without** `stdio.h`

```c
/* we do not need stdio.h */
int printf(const char*, ...);
int scanf(const char*, ...);
int main(){
  char name[80];
  printf("Hello what is your name\n");
  scanf("%79s", name);
  printf("Hello %s!\n", name);
}
```


Linking libraries

- **The body of the function is not in the** `.h` **file**
  - It contains only function prototypes
  - And constant definitions
- **The body has been compiled once**
  - Generated a `.o` file (or something similar)
  - That is linked by `gcc` at linking time
  - Various `.o` files are combined to form one single program

Programs written in multiple files

- **Define functions in different** `.c` **files**
  - Group functions by usage
  - Functions that work together
  - Functions that have a similar topic
  - Functions that you want to reuse in the same area
  - A single file should not be too large
- **For each** `.c` **file, write a** `.h` **file**
  - Contains *exported* function prototypes
  - Contains *exported* constant definitions

Functions that should not be exported must be declared as `static`.

Example : myFirstLibrary.c

```
#include "myFirstLibrary.h"
long factorial(long val){
  long res= 1;
  long i;
  for(i=1; i <= val;i++){
    res *= i;
  }
  return res;
}
long logarithm(long val){
  long res=0;
  if(val<=1){
    return 0;
  }
  while(val>1){
    val /=2;
    res++;
  }
  return res;
}
```


Example : myFirstLibrary.h

- **myFirstLibrary.h**

```
long factorial(long);
long logarithm(long);
```


Use of the library

- **Program using functions from our library**
  - Must include the functions prototypes from `myFirstLibrary.h`
  - Can use functions and constants
  - Do not contain the definition of the functions
- **Example of use: uselibraries.c**

```
#include <stdio.h>
#include "myFirstLibrary.h"

int main(){
  long i;
  puts("Please␣type␣a␣number:");
  scanf("␣%ld",&i);
  long logi = logarithm(i);
  long factoriali = factorial(logi);
  printf("log(%ld)=%ld␣and␣%ld!␣=␣%ld\n",i,↘
  →logi,logi,factoriali);
}
```


Compilation / Linking

- **Each of the files must be compiled separately**
  - Each `.c` is compiled into a `.o` file
  - Contains the bodies of all the functions
- **Linking**
  - Each time a library is used
  - Its `.o` files must be linked with the `.o` file containing the `main()` function.
- **Traditional Makefile**

```
myFirstLibrary.o: myFirstLibrary.c
        gcc myFirstLibrary.c -c

useLibrary.o: useLibrary.c
        gcc useLibrary.c -c

useLibrary: useLibrary.o myFirstLibrary.o
        gcc useLibrary.o myFirstLibrary.o -o↘
        → useLibrary
```

Makefile.am

```
bin_PROGRAMS = \
  allinone \
  withlibrary
allinone_SOURCES = \
  myFirstLibrary.c \
  useLibrary.c
lib_LTLIBRARIES = \
  libmy.la
libmy_la_SOURCES = \
  myFirstLibrary.c
withlibrary_SOURCES = \
  useLibrary.c
withlibrary_LDADD = \
  libmy.la
include_HEADERS = \
  myFirstLibrary.h
```


Scope of variables

Automatic Variables

Automatic variables

- **Default for variables in a function**
  - can be explicitely marked with the word `auto`

```
void hello(void){
  auto long i,j;
  auto double x;
  ...
}
```

is equivalent to

```
void hello(void){
  long i,j;
  double x;
  ...
}
```

Automatic variables(Cont.)

- **Scope**
  - The scope is where the variable can be accessed (read and write)
  - Scope of an automatic variable =
    lexical scope of the statement where it was defined.

Automatic variables(Cont.)

- **Example:**

```
#include <stdio.h>
#define MAX 10
void print_n_stars(int nbStars){
  int i;
  printf("%d)",i);
  for(i=0;i<nbStars;i++){
    printf("*");
  }
  printf("\n");
}
int main(){
  for(int i=0;i<MAX;i++){ // c99
    print_n_stars(i);
  }
}
```


Automatic variables (Cont.)

- **Example with variables in nested statements**

```
include <stdio.h>
#define MAX 8

int main(){
  int i;
  for(i=0; i<MAX; i++){
    if(i<3){
      int i=9;
      printf("i=%d ;",i);
    }
    else{
      printf("i=%d ;",i);
    }
  }
  putchar('\n');
}
/* Output:
i=9 ;i=9 ;i=9 ;i=3 ;i=4 ;i=5 ;i=6 ;i=7 ;
*/
```

Static variables

- **Principle**
  - Static variables are allocated for the duration of the program
  - They are accessible from the scope of their definition
  - Typical scopes are a source file or a function
  - Static variables must not be declared in headers!
  - If an initializer is provided, it is run only once
- **Syntax**

```
static int i = 0;
```


Static variable

- **Example**

```
#include <stdio.h>
int counter(void){
  static int i = 0;
  i++;
  return i;
}
int main(){
  counter();
  counter();
  counter();
  counter();
  counter();
  printf("Counter = %d\n",counter());
  /* Output:  Counter = 6   */
}
```


Static functions

Static functions

- **Static functions are limited in scope to the current compilation unit**
  - Libraries should be careful about which functions are exported
  - Limiting the scope allows using the same function name again
  - Not exporting symbols results in smaller, faster binaries
  - Use `static` for all functions *unless* you intend to call the from another source file
  - Group functions into source files to minimize non-static functions
- **Example**

```
#include <stdio.h>
static void helper(void){
  printf("Hi\n");
}
int main(){
  helper();
  helper();
}
```

Global / Extern

Global / Extern Variables

- **Global Variables are defined outside any function**
  - The can be accessed from within any function in the file
  - Functions defined after the definition of the variable can read and write the variable.
  - Corresponds to the variables in Assembly
- **Example**

```
#include <stdio.h>
int myVar;
static void printVar(void){
  printf("myVar␣=%d\n",myVar);
}
int main(){
  myVar = 10;
  printVar();
  myVar /=2;
  printVar();
  /* Output:   myVar =10
               myVar =5   */
}
```


Extern variables

- **Global variables can be accessed from other files**
  - Storage-class *external* (`extern`)
  - Use `extern` to declare globals in headers
  - Define the global variable *once* without `extern` in a ".c" file
- **Example**

```
extern int myVar; /* usually via #include */
void function1(void){
  myVar = 1;
}
```


Arrays

Arrays

- **C provides a data structure for storing elements of the same type**
  - Access is constant time
  - The first element has index 0
  - Last element index size-1
- **Array is an address in Memory**
  - Array can be seen as a label in memory (like in Assembly)
  - a[0] is the first element (at address a)
  - a[5] is the sixth element (at address `a + 5 * size of one element`).

Declaring Arrays

- **To define an array, you must specify the type of elements and the number of elements (i.e. the length)**
  - `type arrayName[arrayLength]`
  - Example:

    ```
    int myArray[10]
    ```

    Defines an array containing 10 integers
- **Initialize array in one statement**

  Specifying the number of elements

  ```
  double balance[5] = {1000.0, 2.0, 3.4, 7.0, ↘
  →50.0};
  ```

  Or without specifying it

  ```
  double balance[] = {1000.0, 2.0, 3.4, 7.0, ↘
  →50.0};
  ```
- **Or using multiple statements**

  ```
  balance[4] = 50.0;
  ```


Accessing Elements

- **One can access elements with the [] operator**
  - It accesses the memory at address:
    start of the array + index*size

```
#include <stdio.h>
#define SIZEARRAY1 5

int main(){
  double array1[SIZEARRAY1]={1.3, 4.5, ↘
  →8.9,-1.0,3};
  int i;
  for(i=0; i < SIZEARRAY1 ; i++){
    printf("array[%d]=%lf\n",i,array1[i]);
  }
}
```


Multidimentional Arrays

- **C programming language allows multidimensional arrays**
  - Declaration of a multidimensional array:

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

- **Example**

```
int myarray[5][10][4]
```

Two Dimensional Array

- **Definition**
  - int a[3][4]
- **Initialization**

```
int a[3][4] = {
 {0, 1, 2, 3} ,   /*  initializers for row ↘
 →indexed by 0 */
 {4, 5, 6, 7} ,   /*  initializers for row ↘
 →indexed by 1 */
 {8, 9, 10, 11}   /*  initializers for row ↘
 →indexed by 2 */
};
```

Or

```
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

- **Accessing elements**

```
int v = a[1][2]; // is 6
```


Example

```
#include <stdio.h>

int main (){
   /* an array with 5 rows and 2 columns*/
   int a[5][2] = { {0,0}, {1,2}, {2,4}, ↘
   →{3,6},{4,8}};
   int i, j;

   /* output each array element's value */
   for ( i = 0; i < 5; i++ )
   {
      for ( j = 0; j < 2; j++ )
      {
         printf("a[%d][%d] = %d\n", i,j, a[i][j↘
         →] );
      }
   }
   return 0;
}
```


Passing arrays to functions

- **Formal parameters as a sized array**

```
void myFunction(int param[10]){
  ...
}
```

- **Formal parameters as a pointer**

```
void myFunction(int *param){
  ...
}
```

(we will see pointers more in detail next week)

- **Unsized array**

```
void myFunction(int param[]){
  ...
}
```


Example

```
#include <stdio.h>
/* function declaration */
static double
getAverage(int arr[], unsigned int len){
  double sum = 0.0;
  for (unsigned int i = 0; i < len; ++i) // c99
    sum += arr[i];
  return sum / len;
}
int main (){
  int balance[5] = {1000, 2, 3, 17, 50};
  double avg = getAverage(balance, 5);
  printf("Average value is: %f \n", avg);
  return 0;
}
```

Return an array from a function

- **Return value `type*` (pointer on one element)**
  - Corresponds to the address of the first element
  - It is the definition of an array

```
int * myFunction(){
...
}
```

- **Example**

Example

```
#include <stdio.h>
int * getRandom( ){
  static int  r[10];
  srand( (unsigned)time( NULL ) );
  for (int i = 0; i < 10; ++i) { // c99
     r[i] = rand();
     printf( "r[%d] = %d\n", i, r[i]);
  }
  return r;
}
int main (){
   int *p = getRandom();
   for (int i = 0; i < 10; i++ ){ // c99
       printf( "*(p + %d) : %d\n", i, *(p + i));
   }
   return 0;
}
```


Low Level Programming

Low Level Programming

- **Bitwise Operations**
  - Work directly with the bits
  - Equivalent with Assembly instructions
- **Logical operations**
  - Negation (one's complement) : ~
  - AND : &
  - OR: |
  - XOR: ^

One's complement

- **Negate a number bitwise**
  - A number is seen as an array of bits
  - 0xF5=1111 0101
  - ~0xF5 = 0000 1010 = 0x0A

```
int number1 = 0x1;
int number2 = ~number1; // One's ↘
→complement to number1
printf("n1=␣%X␣and␣n2=%X␣\n",number1,↘
→number2);
number1 = 0xF0FF00;
number2 = ~number1; // One's complement to↘
→ number1
printf("n1=␣%X␣and␣n2=%X␣\n",number1,↘
→number2);
 /* Output:
   One's complement (negate each bit)
   n1= 1 and n2=FFFFFFFE
   n1= F0FF00 and n2=FF0F00FF */
```


Binary Operations

- **And, Or , Xor**

```
int n1 = 0xFF11;
int n2 = 0x5599;
int n3 = n1 & n2;
printf("%X␣AND␣%X␣=␣%X␣\n",n1,n2,n3);
int n4 = n1 | n2;
printf("%X␣OR␣%X␣=␣%X␣\n",n1,n2,n4);
int n5 = n1 ^ n2;
printf("%X␣XOR␣%X␣=␣%X␣\n",n1,n2,n5);
/* Output:
  FF11 AND 5599 = 5511
  FF11  OR 5599 = FF99
  FF11 XOR 5599 = AA88
 */
```


Shift

- **Shift the bits of the number**
  - A number is seen as an array of bits
  - Bits are shifted left and right
- **the << operator**
  - << x Shifts x bits to the left
- **the >> operator**
  - >> x Shifts x bits to the right

```
x = y >> 2; /* x = y shifted two times ↘
→to the left */
z = t << 2; /* z = t shifted two times ↘
→to the right */
```


Shift: Example

```
puts("Shift␣to␣the␣left␣3␣bits");
 unsigned int a = 0xFF00;
 unsigned int b = a << 3;
 printf("a=%X;␣b=%X\n",a,b);
 puts("Shift␣to␣the␣right␣3␣bits");
 unsigned int a2 = 0xFF00;
 unsigned int b2 = a >> 3;
 printf("a2=%X;␣b2=%X\n",a2,b2);
```

Shift: Example

```
puts("Shift␣to␣the␣left␣4␣bits");
 a = 0xFF00;
 b = a << 4;
 printf("a=%X;␣b=%X\n",a,b);
 puts("Shift␣to␣the␣right␣4␣bits");
 a2 = 0xFF00;
 b2 = a >> 4;
 printf("a2=%X;␣b2=%X\n",a2,b2);
```


Conclusion

Conclusion

- **Program may be split across multiple files**
  - Files should contain functions that belong together
  - Libraries: even larger aggregations of functions
  - `myprog.h`: declarations of the functions
  - `myprog.c`: definition of the functions
  - `myprog.h`: must be included in all files where it can be used
  - `myprog.o`: must be linked where the functions are used
- **Variables**
  - Automatic: seen in a block (and blocks inside this one)
  - Global / external: seen everywhere (even in other files)
  - Static: initialized only one for an execution of the program (even if the function is called many times)

Conclusion

- **Arrays**
  - Structure for storing in memory a collection of informations of the same type.
- **Low level programming**
  - Possibility to work at the bit level
  - Shift, logical bitwise operators (AND, OR, XOR, NOT)

Bibliography

- This course corresponds to chapters *7*, *8* and *9* of the course book:
  Schaum's OuTlines, Programming with C (second edition), *Byron Gottfied*, Mc Graw-Hill, 1996
- **Web site[1]:** `http://www.tutorialspoint.com/cprogramming/c_arrays.htm`

[1] Visited on November 27, 2014