Chapter 5. Functions

Byoung-Tak Zhang
TA: Hanock Kwak

Biointelligence Laboratory
School of Computer Science and Engineering
Seoul National Univertisy

http://bi.snu.ac.kr

Functions

- The heart of effective problem solving is **problem decomposition**.
  - breaking a problem into small, manageable pieces
  - In C, the functions are those pieces.
- A program consists of one or more files
  - Each file contains zero or more functions, one of them being a **main()** function.

Function Definition

```
return_type  function_name ( parameter_list )   : function header
{
   declarations                 : function body
   statements
}
```

- A function definition starts with the ***return_type*** of the function.
  - If no value is returned, then the ***return_type*** is **void**.
  - If NOT **void**, then the value returned by the function will be converted, if necessary, to this ***return_type*** .
- ***parameter_list***
  - a comma-separated list of declarations
  - formal parameters of the function

Function Definition

```
int factorial(int n)                    /* header */
{                                       /* body starts here */
    int i, product = 1;
    for (i = 2; i <= n; ++i)
        product *= i;
    return product;
}

void main(void)
{
    …
    factorial(7);     /* The function, factorial(), is called, or invoked */
    …

}
```

Function Definition

```
void nothing(void) { } /* this function does nothing */

double twice(double x)
{
   return 2.0 * x;
}

int add_all(int a, int b)
{
   return (a+b);
}
```

Function Definition

- "local" variables vs. "global" variables
  - "local" variables : any variables declared in the body of a function
  - "global" variables : other variables declared external to the function

```
#include <stdio.h>

int a = 33; // a is external, it is accessible from all functions in this file

int main(void) {
    int b = 77; /* b is local to main() */
    printf("a = %d\n", a); /* a is global to main() */
    printf("b = %d\n", b);
    return 0;
}
```

Function Definition

- Important reasons to write programs as collections of many small functions
  - It is simpler to correctly write a small function to do one job.
    - easier writing & debugging
  - It is easier to maintain or modify such a program
  - Small functions tend to be self-documenting and highly readable.

return Statement

return;

return *expression*;

- When a **return** statement is encountered,
  - execution of the function is terminated and
  - control is passed back to the calling environment

< Examples >

```
return;
return ++a;
return (a*b);
```

return Statement

```
float f(char a, char b, char c)
{
    int i;
    …
    return i; /*value returned will be converted to a float*/
}

double absolute_value(double x)
{
    if (x>=0.0)
        return x;
    else
        return -x;
}
```

Function Prototypes

```
return_type  function_name( parameter_type_list ) ;
```

- Functions should be declared before they are used.
- Function prototype
  - tells the compiler the number and type of argument passed to the function
  - tells the type of the value returned by the function
  - allows the compiler to check the code more thoroughly

< Examples >

```
double sum(double, double);
int absolute_value( int );
float area( float, float );
```

< Identifiers are optional >

```
double sum(double a, double b);
int absolute_value(int value);
float area(float width, float height);
```

How to code functions (single file)

```
#include <stdio.h>

// declare function prototypes before main function
int sum(int, int);
int abs(int);

int main() // you can call declared functions
{
    int a, b, c;
    scanf("%d %d", &a, &b);
    printf("|a| + |b| = %d \n", sum( abs(a), abs(b) ) );
    return 0;
}
// continues to next page !!
```

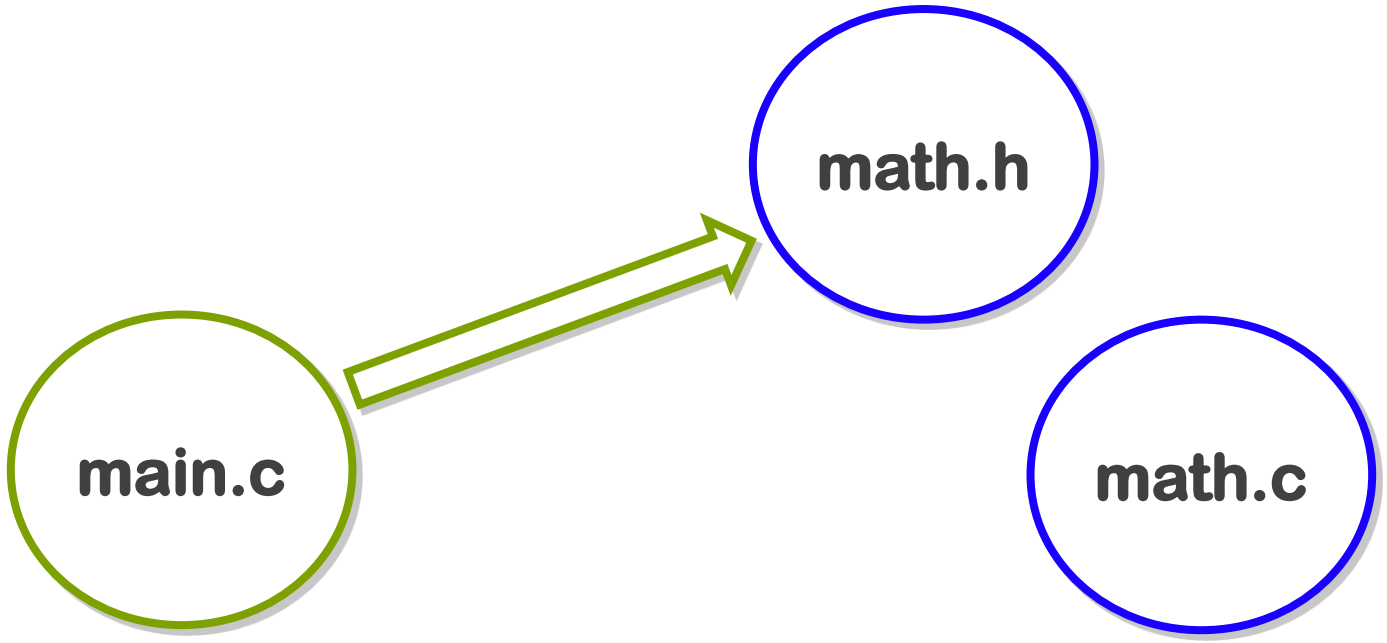
How to code functions (single file)

```
// write definitions of functions declared above
int sum(int a, int b)
{
   return a + b;
}

int abs(int a)
{
   if (a >= 0)
      return a;
   else
      return -a;
}
```

How to code functions (multiple file)

- Writing a program in a single source file is messy.
- Organization
  - Collect closely related functions for each source file.
  - Create a header file for each source file and write function prototypes in it.
  - Include external header file to use functions in it.



How to code functions (multiple file)

- **math.h**

```
// declare function prototypes in the header file
int sum(int, int);
int abs(int);
```

How to code functions (multiple file)

- **math.c**

```
// include math.h to link definitions with prototypes
#include "math.h"

int sum(int a, int b)
{
   return a + b;
}

int abs(int a)
{
   if (a >= 0)
      return a;
   else
      return -a;
}
```

How to code functions (multiple file)

- **main.c**

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

int main()
{
    int a, b, c;
    scanf("%d %d", &a, &b);
    printf("|a| + |b| = %d \n", sum( abs(a), abs(b) ) );
    return 0;
}
```

Scope Rules

- Basic rules of scoping
  - Identifiers are accessible only within the block in which they are declared.
  - They are unknown outside the boundaries of that block.
  - When an inner declared identifier has same name as outer one, it hides the outer one.
- Why blocks(compound statements)?
  - to allow memory for variables to be allocated where needed
  - Block exit releases the allocated storage.

Scope Rules

```
{
    int a = 1, b = 2, c = 3;
    printf("%d %d %d\n", a, b, c); /* 1 2 3 */
    {
        int b = 4;
        float c = 5.0;
        printf("%d %d %.1f\n", a, b, c); /* 1 4 5.0 */
        a = b;
        {
            int c;
            c = b;
            printf("%d %d %d\n", a, b, c); /* 4 4 4 */
        }
        printf("%d %d %.1f\n", a, b, c); /* 4 4 5.0 */
    }
    printf("%d %d %d\n", a, b, c); /* 4 2 3 */
}
```

Storage Classes

- Every variable and function in C has two attributes
  - *type* and *storage class*

- Four storage classes
  - **auto extern register static**
  - **auto**: automatic
    - The most common storage class for variable

Storage Class auto

- Variables declared within function bodies are automatic by default
  - When a block is entered, the system allocates memory for the automatic variables.
    - These variables are “local” to the block
  - When the block is exited, the memory is released (the value is lost)

```
void f(int m)
{
    int a,b,c;
    float f;
    ….
}
```

Storage Class extern

- One method of transmitting information across blocks and functions is **to use external variables**
- When a variable is declared outside a function,
  - storage is permanently assigned to it.
  - its storage class is **extern**
  - The variable is “global” to all functions declared **after** it.
- Information can be passed into a function two ways
  - by use of external variables
  - by use of the parameter mechanism

Storage Class register

- The storage class **register**
  - tells the compiler that the associated variables should be stored in high-speed memory registers
  - aims to improve execution speed
    - declares variables most frequently accessed as **register**

```
{
    register int i;        /* register i */
    for (i = 0; i < LIMIT; i++) {
    …
    }
} /* block exit will free the register */
```

Storage Class static

- The storage class **static**
  - allows a local variable to retain its previous value when the block is reentered.
  - in contrast to ordinary **auto** variables

```
void f();

void main()
{
    f();
    f();
    f();
}
```

```
void f()
{
    static int cnt = 0;
    printf("%d\n", cnt);
    cnt++;
}
```

Default Initialization

- **external** and **static** variables
  - initialized to zero by the system, if not explicitly initialized by programmers

- **auto** and **register** variables
  - usually not initialized by the system
  - have “garbage” values.

Recursion

- A function is recursive if it calls itself, either directly or indirectly

```
#include <stdio.h>
int sum(int n);
int main()
{
    int n = 5;
    printf("The sum from 1 to %d is %d\n", n, sum(n));
    return 0;
}
int sum(int n)
{
    return (n <= 1) ? n : n + sum(n-1);
}
```