

CSE 230
Intermediate Programming
in C and C++
Functions

Fall 2017

Stony Brook University

Instructor: **Shebuti Rayana**

shebuti.rayana@stonybrook.edu

<http://www3.cs.stonybrook.edu/~cse230/>

Concept of Functions in C

- The heart of effective problem solving is problem decomposition. Taking a problem and breaking it into small, manageable pieces is critical to writing large programs.
- In C, the **function** construct is used to implement this "top-down" method of programming.
- A program consists of one or more files, each file containing zero or more functions, one of them being a `main()` function.
- Functions are defined as individual objects that cannot be nested.
- Program execution begins with `main()`, which can call other functions, including library functions such as `printf()` and `scanf()`.
- Functions operate with program variables, and which of these variables is available at a particular place in a function is determined by scope rules.

Why Functions?

- There are several reasons to write programs as collections of many functions. It is simpler to correctly write a small function to do one job.
- Both the writing and debugging are made easier.
- It is also easier to maintain or modify such a program. One can readily change just the set of functions that need to be rewritten, expecting the rest of the code to work correctly.
- Small functions tend to be self documenting and highly readable.

Overview

- Function definition
- Function declaration
- Scope rules
- Storage classes
- Recursion

Function Definition

- The C code that describes what a function does is called the function definition. It must not be confused with the function declaration.

- **General Form**

```
return_type function_name(parameter list) {  
    declarations statements }
```

- Everything before braces comprise *header* of the function definition
- Everything between braces comprise the *body* of the function definition
- Parameter list is comma separated list of declarations

Example 1: Function Definition

```
1  int factorial(int n) /*function header*/
2  {                    /*function body starts here*/
3      int i, product = 1;
4      for(i=2;i<=n;i++)
5      {
6          product *= i;
7      }
8      return product;
9  }
```

- The first `int` tells the compiler that the value returned by the function will be converted, if necessary, to an `int`.
- The parameter list consists of the declaration `int n`. This tells the compiler that the function takes a single argument of type `int`.
- An expression such as `factorial(7)` causes the function to be invoked, or called.
- The effect is to execute the code that comprises the function definition, with `n` having the value 7.
- Finally, the function will return the factorial value of 7.

Example 2: Function Definition

```
1 void wrt_address(void)
2 {
3     printf("%s\n%s\n%s\n%s\n%s\n\n",
4           "*****",
5           "**  SANTA CLAUS  **",
6           "**  NORTH POLE  **",
7           "**  EARTH      **",
8           "*****");
9 }
```

- The first void tells the compiler that this function returns no value; the second void tells the compiler that this function takes no arguments.
- Following expression causes the function to be invoked.

```
wrt_address();
```

- For example, to call the function three times we can write

```
int i;
for (i = 0; i < 3; i++)
    wrt_address();
```

Function Definition (cont.)

- If no return type is specified, then it is `int` by default
- Any variables declared in the body of a function are called *local* variables to that function.
- Other variables declared external to the function are called *global* variables.

```
1  #include <stdio.h>
2
3  int a = 33; /* a is external and initialized to 33 */
4
5  int main(void)
6  {
7      int b = 77; /* b is local to main() */
8
9      printf("a = %d\n",a); /* a is global to main() */
10     printf("b = %d\n",b);
11     return 0;
12 }
```


The return statement

- The return statement may or may not contain an expression
`return;` OR `return expression;`
- Example:

```
return; return 1; return ++a; return (a*b);
```

- When a return statement is encountered, execution of the function is terminated and control is passed to the caller.
- If the return contains an expression, then the value of the expression is passed to the caller as well.
- There can be zero or more return statements in a function.
- When there is not return, control passed to the caller when the closing brace is encountered.

Function prototype

- A function should be declared before it is used.
- C provides a function declaration syntax called **function prototype**
- A function prototype tells the compiler the number and type of arguments that are to be passed to the function and the type of the value that is to be returned by the function.
- **Example:** `double sqrt(double);`
- **General:** `type function_name(parameter type list);`

Example: Function Prototype

```
1  #include <stdio.h>
2
3  void print_header(void);
4  void print_table(int);
5  long power(int, int);
6
7  int main()
8  {
9      int N = 7;
10     print_header();
11     print_table(N);
12     return 0;
13 }
14
15 void print_header(void)
16 {
17     printf("----Print Table of Powers----\n");
18 }
19
20 void print_table(int n)
21 {
22     int i, j;
23     for(i=1;i<=n;i++)
24     {
25         for(j=1;j<=n;j++)
26             printf("%ld ",power(i,j));
27         putchar('\n');
28     }
29 }
30
31 long power(int m, int n)
32 {
33     int i;
34     long product = 1;
35     for(i=1;i<=n;i++)
36         product *= m;
37     return product;
38 }
```

Overview

- Function definition
- Function declaration
- Scope rules
- Storage classes
- Recursion

Function Declaration

- From the compilers viewpoint, function declaration can be generated in various ways. By
 - function invocation
 - function definition
 - explicit function declaration
 - function prototype

If a function call, say $f(x)$, is encountered before any declaration, definition, or prototype for it occurs, then the compiler assumes a default declaration of the form `int f()`

```
#include <stdio.h>

int main (void)
{
    printf("%d", sum(10, 5));
    return 0;
}

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

This code compiles in gcc but returns garbage value

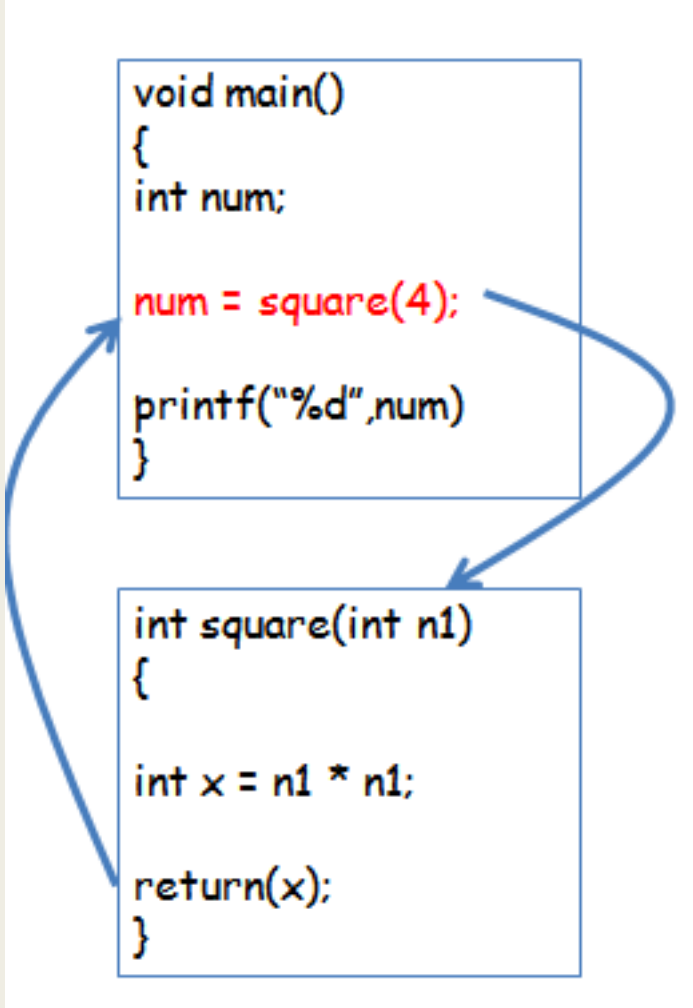
Alternate Style of Function Definition

```
1 #include <stdio.h>
2
3 long power(int m, int n)
4 {
5     int i;
6     long product = 1;
7     for(i=1;i<=n;i++)
8         product *= m;
9     return product;
10 }
11
12 void print_header(void)
13 {
14     printf("----Print Table of Powers----\n");
15 }
16
17 void print_table(int n)
18 {
19     int i, j;
20     for(i=1;i<=n;i++)
21     {
22         for(j=1;j<=n;j++)
23             printf("%ld ",power(i,j));
24         putchar('\n');
25     }
26 }
27
28 int main()
29 {
30     int N = 7;
31     print_header();
32     print_table(N);
33     return 0;
34 }
```

Function Invocation

- Program execution always begins with `main()`.
- When program control encounters a function name, the function is called, or invoked. This means that program control passes to that function.
- After the function does its work, program control is passed back to the calling environment, which then continues with its work.

```
void main()
{
  int num;
  num = square(4);
  printf("%d",num)
}
```



```
int square(int n1)
{
  int x = n1 * n1;
  return(x);
}
```

Call-by-value

- If a function is “called by value”
 - This means that each argument is evaluated, and its value is used locally in place of the corresponding formal parameter.
 - Thus, if a variable is passed to a function, the stored value of that variable in the calling environment will not be changed.

Example: Call-by-value

```
1  #include <stdio.h>
2
3  int compute_sum(int n);
4
5  int main(void)
6  {
7      int n = 3, sum;
8      printf("%d\n", n); /* 3 is printed */
9      sum = compute_sum(n);
10     printf("%d\n", n); /* 3 is printed */
11     printf("%d\n", sum); /* 6 is printed */
12     return 0;
13 }
14
15 int compute_sum(int n) /*sum the integers from 1 to n*/
16 {
17     int sum = 0;
18     for(; n>0; --n) /*stored value of n is changes*/
19         sum += n;
20     return sum;
21 }
```

Call-by-reference

- "call-by-reference" is a way of passing addresses (references) of variables to a function that then allows the body of the function to make changes to the values of variables in the calling environment.

Call by value

```
1 #include <stdio.h>
2
3 void swap(int i, int j)
4 {
5     int temp;
6     temp = i;
7     i = j;
8     j = temp;
9 }
10
11 int main()
12 {
13     int i = 5;
14     int j = 10;
15     swap(i,j);
16     printf("i = %d\n",i);
17     printf("j = %d\n",j);
18 }
```

Output:
i = 5
j = 10

```
1 #include <stdio.h>
2
3 void swap(int *i, int *j)
4 {
5     int temp;
6     temp = *i;
7     *i = *j;
8     *j = temp;
9 }
10
11 int main()
12 {
13     int i = 5;
14     int j = 10;
15     swap(&i,&j);
16     printf("i = %d\n",i);
17     printf("j = %d\n",j);
18 }
```

Output:
i = 10
j = 5

Call by reference

Overview

- Function definition
- Function declaration
- Scope rules
- Storage classes
- Recursion

Scope Rules

- The basic rule of scoping is that identifiers are accessible only within the block in which they are declared.
- They are unknown outside the boundaries of that block.

```
1 {  
2     int a = 2;           /* outer block a */  
3     printf("%d\n", a);  /* 2 is printed */  
4     {  
5         int a = 5;      /* inner block a */  
6         printf("%d\n", a); /* 5 is printed */  
7     }                  /* back to the outer block */  
8     printf("%d\n", ++a); /* 3 is printed */  
9 }
```

Scope Rules for Nested Blocks

```
3 ▾ {
4     int a = 1, b = 2, c = 3;
5     printf("%d %d %d\n", a, b, c);           /* 1 2 3 */
6 ▾     {
7         int b = 4;
8         float c = 5.0;
9         printf("%d %d %f\n", a, b, c);       /* 1 4 5.0 */
10        a = b;
11 ▾        {
12            int c;
13            c = b;
14            printf("%d %d %d\n", a, b, c);    /* 4 4 4 */
15        }
16        printf("%d %d %f\n", a, b, c);       /* 4 4 5.0 */
17    }
18    printf("%d %d %d\n", a, b, c);           /* 4 2 3 */
19 }
```

Overview

- Function definition
- Function declaration
- Scope rules
- Storage classes
- Recursion

Storage Classes

- Every variable and function in C has two attributes:
 - *type*
 - *storage class*
- The four storage classes are automatic, external, register, and static
- **Specifiers:** `auto`, `extern`, `register`, `static`
- Most common storage class is automatic

Storage class: `auto`

- Variables declared within function bodies are automatic by default
- Variables defined at the beginning of a block are also automatic by default
- We can also explicitly specify automatic variables
`auto int a;`
- When execution enters a block, system allocates memory for the automatic variables
- When execution exits the block, system releases the memory allocated for automatic variables (so the values are lost)

Storage class: `extern`

- One method of transmitting information across blocks and functions is to use external variables.
- When a variable is declared outside of a function, storage is permanently assigned to it, and the storage class is `extern`.

- `extern` variable is considered global for all the functions declared after it.
- Never disappears

```
#include <stdio.h>

int    a = 1, b = 2, c = 3;          /* global variables */
int    f(void);                    /* function prototype */

int main(void)
{
    printf("%3d\n", f());           /* 12 is printed */
    printf("%3d%3d%3d\n", a, b, c); /* 4 2 3 is printed */
    return 0;
}

int f(void)
{
    int    b, c;                    /* b and c are local */
                                           /* global b, c are masked */
    a = b = c = 4;
    return (a + b + c);
}
```

Storage class: `extern`

In file `file1.c`

```
#include <stdio.h>

int    a = 1, b = 2, c = 3;    /* external variables */
int    f(void);

int main(void)
{
    printf("%3d\n", f());
    printf("%3d%3d%3d\n", a, b, c);
    return 0;
}
```

In file `file2.c`

```
int f(void)
{
    extern int    a;            /* look for it elsewhere */
    int          b, c;

    a = b = c = 4;
    return (a + b + c); }
}
```

The keyword `extern` is used to tell the compiler to look for it elsewhere, either in this file or in some other file.

Storage class: `register`

- Tells the compiler to store the associated variables in high speed memory registers.
- Defaults to automatic if compiler is unable to store (resource limitations) the variable into a high-speed memory register.
- It is used to improve execution speed.
- When speed is a concern, programmer may choose a few variables that are frequently accessed and declare them to be of storage class `register`.

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

Storage class: `register`

- If a storage class `register` is specified in a declaration and the type is absent, then the type `int` by default.
- Note that in our example the register variable `i` was declared as close to its place of use as possible. This is to allow maximum availability of the physical registers, only when needed.
- Always remember that a register declaration is taken only as advice to the compiler.

Storage class: `static`

- Storage class `static` has two distinct usage
 1. The more elementary use is to allow a local variable to retain its previous value when the block is reentered.
- This is contrast to ordinary automatic variables, which lose their value upon block exit and must be reinitialized.

Example

```
void f(void)
{
    static int cnt = 0;
    ++cnt;
    if(cnt%2 == 0)
        // do something
    else
        // do something
}
```

The first time the function is invoked, the variable `cnt` is initialized to zero. On function exit, the value of `cnt` is preserved in memory. Whenever the function is invoked again, `cnt` is not reinitialized.

Static External Variable

- The second and more subtle use of `static` is in connection with external declarations.
 - They are scope restricted external variables.
 - The scope is the remainder of the source file in which they are declared.
- Thus, they are unavailable to functions defined earlier in the file or to functions defined in other files, even if these functions attempt to use the `extern` storage class keyword.

```
void f(void)
{
    /* v is not available here */
}

static int v; /* static external variable */

void g(void)
{
    /* v is available here */
}
```

Example: Static External Variable

```
/* A family of pseudo random number generators. */
#define INITIAL_SEED      17
#define MULTIPLIER        25173
#define INCREMENT         13849
#define MODULUS           65536
#define FLOATING_MODULUS 65536.0

static unsigned seed = INITIAL_SEED; /* external, but */
                                     /* private to this file */

unsigned random(void)
{
    seed = (MULTIPLIER * seed + INCREMENT) % MODULUS;
    return seed; /* return integer between 0 and MODULUS */
}

unsigned probability(void)
{
    seed = (MULTIPLIER * seed + INCREMENT) % MODULUS;
    return seed/FLOATING_MODULUS; /* return float between 0 and 1 */
}
```

Default Initialization

- In C, both external variables and static variables that are not initialized by the programmer are initialized to zero by the system.

Static Functions

- This causes scope of the function to be restricted
- The static functions are only visible within the file in which they are defined
- They can not be accessed from other files

Limitations of Function Definition and Prototypes

- The function storage class specifier, if present, can be either `extern` or `static`, but not both
- `auto` and `register` cannot be used
- The types "array of... " and "function returning ... " cannot be returned by a function. However, a pointer representing an array or a function can be returned.
- The only storage class specifier that can occur in the parameter type list is `register`.

Overview

- Function definition
- Function declaration
- Scope rules
- Storage classes
- Recursion

In next class