

FUNCTIONS

Erkut ERDEM
Hacettepe University
October 2010

Introduction

Structured Programming is a problem-solving strategy and a programming methodology that includes the following two guidelines:

- The flow of control in a program should be as simple as possible.
- The construction of a program should embody top-down design.

Top-down Design

Top-down design, stepwise refinement, or divide and conquer

- consists of repeatedly decomposing a problem into smaller problems.
 - i.e., construct a program from smaller pieces (components, modules)
- Each piece more manageable than the original program
- Create new primitives

Functions

- Functions
 - Programs combine user-defined functions with library functions
 - C standard library has a wide variety of functions
- Function calls
 - Invoking functions
 - Provide function name and arguments (data)
 - Function performs operations or manipulations
 - Function returns results
 - Function call analogy:
 - Boss asks worker to complete task
 - Worker gets information, does task, returns result
 - Information hiding: boss does not know details

Functions

- We have already written our own functions and used library functions:
 - `main` is a function that must exist in every C program.
 - `printf`, `scanf` are library functions which we have already used in our programs.
- We need to do two things with functions:
 - create functions
 - call functions (Function invocation)

Function Definition

A function definition has the following form:

```
return_type function_name (parameter-declarations)  
{  
    variable-declarations  
    function-statements  
}
```

return_type - specifies the type of the function and corresponds to the type of value returned by the function

- `void` – indicates that the function returns nothing.
- if not specified, of type `int`

function_name – name of the function being defined (any valid identifier)

parameter-declarations – specify the types and names of the parameters (a.k.a. formal parameters) of the function, separated by commas.

Example: Function returning a value

- Let's define a function to compute the cube of a number:

```
int cube ( int num ) {  
    int result;  
  
    result = num * num * num;  
    return result;  
}
```

- This function can be called as:

```
n = cube(5);
```

Example: void Function

```
void prn_message(void) /* function definition */  
{  
    printf("A message for you:  ");  
    printf("Have a nice day!\n");  
}  
  
int main (void)  
{  
    prn_message ( );      /* function invocation */  
    return 0;  
}
```

Math Library Functions

- Math library functions
 - perform common mathematical calculations
 - `#include <math.h>`
- Format for calling functions
 - `FunctionName ( argument ) ;`
 - If multiple arguments, use comma-separated list
 - `y = sqrt( 900.0 );`
 - Calls function `sqrt`, which returns the square root of its argument
 - Arguments may be any r-value (constants, variables, or expressions)

Math Library Functions

<u>Function Header</u>	<u>Description</u>
<code>int abs(int num)</code>	Returns the absolute value of an integer element.
<code>double fabs(double num)</code>	Returns the absolute value of a double precision element.
<code>double pow(double x, double y)</code>	Returns x raised to the power of y.
<code>int rand(void)</code>	returns a random number
<code>double sin(double angle)</code>	Returns the sine of an angle; the angle should be in Radius.
<code>double cos(double angle)</code>	Returns the cosine of an angle; the angle should be in Radius.
<code>double sqrt(double num)</code>	Returns the the square root of a double

Math Library Functions

- Calculate the square root of $(x1 - x2)^2 + (y1 - y2)^2$

```
a = x1 - x2;
b = y1 - y2;
c = pow(a,2) + pow(b, 2);
d = sqrt(d);
```

General Structure of a C Program

```
preprocessor directives

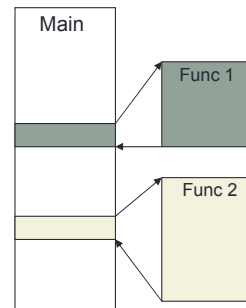
function prototypes

int main () {
.
.
.
}

function-1
function-2
.
.
.
function-n
```

Function Invocation

- A program is made up of one or more functions, one of them being `main()`.
- When a program encounters a function, the function is called or invoked.
- After the function does its work, program control is passed back to the calling environment, where program execution continues.



Variable Declarations within Function Definitions

- Variables declared local to a function supersede any identically named variables outside the function

```
int lcm(int m, int n) {
    int i;
    ...
}

int gcd(int m, int n) {
    int i;
    ...
}
```

The `return` statement

- When a return statement is executed, the execution of the function is terminated and the program control is immediately passed back to the calling environment.
- If an expression follows the keyword `return`, the value of the expression is returned to the calling environment as well.
- A return statement can be one of the following two forms:

```
return;
return expression;
```

Examples

```
return;
```

```
return 1.5;
```

```
return result;
```

```
return a+b*c;
```

```
return x < y ? x : y;
```

Example

```
int IsLeapYear(int year)
{
    return ( (year % 4 == 0) && (year % 100 != 0))
           || (year % 400 == 0) );
}
```

- This function may be called as:

```
if (IsLeapYear(2005))
    printf("29 days in February.\n");
else
    printf("28 days in February.\n");
```

Example

```
#include <stdio.h>
int min(int a, int b)
{
    if (a < b)
        return a;
    else
        return b;
}

int main (void)
{
    int j, k, m;

    printf("Input two integers:   ");
    scanf("%d %d", &j, &k);
    m = min(j,k);
    printf("\nThe minimum is %d.\n", m);
    return 0;
}
```

Input two integers: 5 6
The minimum is 5.

Input two integers: 11 3
The minimum is 3.

Parameters

- A function can have zero or more parameters.
- In declaration header:

```
int f (int x, double y, char c);
```


the *formal parameter list*
(parameter variables and
their types are declared here)

- In function calling:

```
value = f(age, 100*score, initial);
```


actual parameter list
(cannot tell what their type
are from here)  r-value

Rules for Parameter Lists

- The number of parameters in the actual and formal parameter lists must be *consistent*
- Parameter association is *positional*: the first *actual* parameter matches the first *formal* parameter, the second matches the second, and so on
- *Actual* parameters and *formal* parameters must be of compatible *data types*
- *Actual* parameters may be a variable, constant, any expression matching the type of the corresponding formal parameter

Invocation and Call-by-Value

- Each argument is evaluated, and its value is used locally in place of the corresponding formal parameter.
- If a variable is passed to a function, the stored value of that variable in the calling environment will not be changed.
- In C, all calls are call-by-value unless specified otherwise.

Function Call

- The type of a function-call expression is the same as the type function being called, and its value is the value returned by the function.
- Function calls can be embedded in other function calls.

• e.g.

```
t = cubesum(i);  
j = cubesum(t);
```

is equivalent to

```
j = cubesum(cubesum(i));
```

Example

```
#include <stdio.h>

int compute_sum (int n)
{
    int sum;

    sum = 0;

    for ( ; n > 0; --n)
        sum += n;
    printf("%d\n", n);
    return sum;
}

int main (void)
{
    int n, sum;

    n = 3;

    printf("%d\n", n);
    sum=compute_sum(n);
    printf("%d\n", sum);
    return 0;
}
```

3
0
3
6

Example

```
/* Finding the maximum of three integers */
#include <stdio.h>
/* Function maximum definition */
int maximum( int x, int y, int z )
{
    int max = x;

    if ( y > max )
        max = y;

    if ( z > max )
        max = z;

    return max;
}

int main()
{
    int a, b, c;

    printf( "Enter three integers: " );
    scanf( "%d%d%d", &a, &b, &c );
    printf( "Maximum is: %d\n", maximum( a, b, c ) );

    return 0;
}
```

Enter three integers: 22 85 17
Maximum is: 85

Function Call

- ANSI-C does not set the arguments evaluation order in function calls

- **e.g.**

```
x = 7; a = 2.25;
```

```
f(x=6, x-7, a)      f(x=6, x-7, a)
  ↓   ↓   ↓         ↓   ↓   ↓
  6   -1  2.25      6   0  2.25
```

```
printf("%d %d%, i, i++);
printf("%d %d%, i, ++i);    ambiguous
```

Function Prototypes

- General form for a function prototype declaration:
return_type function_name (parameter-type-list)
- Used to validate functions
 - Prototype only needed if function definition comes after use in program
- The function with the prototype

```
int maximum( int, int, int );
```

 - Takes in 3 ints
 - Returns an int

Alternative styles for function definition order

```
#include <stdio.h>

int max(int,int);
int min(int,int);

int main(void)
{
    min(x,y);
    max(u,v);
    ...
}

int max (int a, int b)
{
    ...
}

int min (int a, int b)
{
    ...
}
```

```
#include <stdio.h>

int max (int a, int b)
{
    ...
}

int min (int a, int b)
{
    ...
}

int main(void)
{
    ...
    min(x,y);
    max(u,v);
    ...
}
```

Block Structure

- A block is a sequence of variable declarations and statements enclosed within braces.
- Block structure and the scope of a variable

```
int factorial(int n)
{
    if (n<0) return -1;
    else if (n==0) return 1;
    else
    {
        int i, result=1;
        for (i=1;i<=n; i++) result *= i;
        return result;
    }
}
```

External Variables

- Local variables can only be accessed in the function in which they are defined.
- If a variable is defined outside any function at the same level as function definitions, it is available to all the functions defined below in the same source file
→ external variable
- **Global variables:** external variables defined before any function definition
 - Their scope will be the whole program

Example

```
#include <stdio.h>
void print_message (int k); /*function prototype */

int main (void)
{
    int n;

    printf("There is a message for you.\n");
    printf("How many times do you want to see it? ");
    scanf("%d", &n);
    print_message(n);
    return 0;
}

void print_message (int k) /* function definition */
{
    int i;
    printf("\nHere is the message.\n");
    for (i=0; i < k; ++i)
        printf("Have a nice day!\n");
}
```

Example

```
/* An example demonstrating local variables */
#include <stdio.h>

void func1 (void);

int main (void)
{
    int i = 5;
    printf("%d \n", i);
    func1( );
    printf("%d \n", i);
    return 0;
}

void func1 (void)
{
    int i = 5;
    printf("%d\n", i);
    i++;
    printf("%d\n", i);
}
```

5
5
6
5

Example: Transforming rectangular coordinates to polar coordinates

```
#include <math.h>
#include <stdio.h>
#define PI 3.1415927

float r, theta;
void polar (float x, float y)

int main(void){
    float x, y;
    scanf("%f %f", &x, &y);
    polar(x,y);
    printf("r = %f, theta = %f\n", r, theta);
    return 0;
}

void polar(float x, float y)
{
    if (x==0 && y==0) r = theta = 0;
    else {
        r = sqrt(x*x + y*y);
        theta = atan2(y,x); }
}
```


Static Variables

- A variable is said to be static if it is allocated storage at the beginning of the program execution and the storage remains allocated until the program execution terminates.
- External variables are always static
- Within a block, a variable can be specified to be static by using the keyword static before its type declaration:
`static type variable-name;`
- Variable declared static can be initialized only with constant expressions (if not, its default value is zero)

Example

```
#include <stdio.h>
void incr(void);

int main(void)
{
    int i;
    void incr(void);

    for (i=0; i<3; i++)
        incr();
    return 0;
}

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

Output:

```
static_i = 0
static_i = 1
static_i = 2
```

Example

```
#include <stdio.h>

put_stars(int n)
{
    static int eski_n;
    int i;
    for (i=0; i<eski_n; i++)
        printf(" ");
    for (i=0; i<n; i++)
        printf("*");
    printf("\n");
    eski_n += n;
}

int main(void)
{
    put_stars(3); put_stars(2); put_stars(3);
    return 0;
}
```

Output:

```
***
  **
***
```

Correct the errors in the following program segments

1. `int g (void) {`
 `printf ("Inside function g\n");`

 `int h(void) {`
 `printf("Inside function h\n");`
 `}`
 `}`
2. `int sum(int x, int y) {`
 `int result;`
 `result = x + y;`
 `}`

Correct the errors in the following program segments

```
3. void f (float a); {  
    float a;  
    printf ("%f", a); }
```

```
4. void product (void) {  
    int a, b, c, result;  
    printf("Enter 3 integers: ");  
    scanf("%d %d %d", &a, &b, &c);  
    result = a * b * c;  
    printf("Result is %d\n", result);  
    return result;  
}
```

Exercises

- Define a function to calculate $(x^2 + y^2 + z^2)^{1/2}$ and use it to calculate

$$a = 1/(u^2+v^2+w^2)^{1/2}, \quad b = (u^4 + v^4 + w^4)^{1/2},$$
$$g = (4u^2+9v^2+25w^2)^{1/2}, \quad h = (3u^2)^{1/2}(12v^2)^{1/2}(27w^2)^{1/2}$$

Exercises

- Analyze the output of the following program

```
#include <stdio.h>                int main(void)  
int i=0;                          {  
                                  {  
void f(void)                      int i=4;  
{                                printf("%d\n", i);  
    int i;                       }  
    i = 1;                      printf("%d\n", i);  
}                                f();  
void g(void)                     printf("%d\n", i);  
{                                g();  
    i=2;                        printf("%d\n", i);  
}                                h(i);  
void h(int i)                   printf("%d\n", i);  
{                                return 0;  
    i=3;                        }  
}
```

Programming Exercise

- Write a program that reads in the side of a square and then prints a hollow square. Your program should work for squares of all side sizes between 1 and 20. For example, if your program reads a size of 4, it should print:

```
****  
*  *  
*  *  
****
```

Programming Exercise

- Twin primes are defined to be two consecutive odd numbers which are both primes. For example, 11 and 13 are twin primes. Write a program to generate all the twin primes in a given range of numbers.