CSE123

Program Design
and
Modular Programming—Functions
5.1 Introduction

A function in C is a small “sub-program”
- performs a particular task,
- supports the concept of modular programming design techniques.

In modular programming the various tasks are assigned to individual functions and the main program basically calls these functions in a certain order.
5.1 Introduction

• We have already been exposed to functions.
• The main body of a C program, identified by the keyword `main`, and enclosed by left and right braces is a function.
  ```c
  int main(void)
  {
  ....
  }
  ```
  
• It is called by the operating system when the program is loaded, and when terminated, returns to the operating system.
5.2 Program Modules in C

- Functions
  - Modules in C
  - 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
5.2 Program Modules in C

Fig. 5.1 Hierarchical boss function/worker function relationship.
5.3 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
  - printf( "%.2f", sqrt( 900.0 ) );
    - Calls function sqrt, which returns the square root of its argument
    - All math functions return data type double
  - Arguments may be constants, variables, or expressions
5.3 Math Library Functions

- The following code fragment uses the Pythagorean theorem $c^2 = a^2 + b^2$ to calculate the length of the hypotenuse given the other two sides of a right triangle:

```c
double c, a, b;
c = sqrt(pow(a, 2) + pow(b, 2));
```
## 5.3 Math Library Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>sqrt( x )</td>
<td>square root of x</td>
<td>sqrt( 900.0 ) is 30.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sqrt( 9.0 ) is 3.0</td>
</tr>
<tr>
<td>exp( x )</td>
<td>exponential function $e^x$</td>
<td>exp( 1.0 ) is 2.718282</td>
</tr>
<tr>
<td></td>
<td></td>
<td>exp( 2.0 ) is 7.389056</td>
</tr>
<tr>
<td>log( x )</td>
<td>natural logarithm of x (base $e$)</td>
<td>log( 2.718282 ) is 1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>log( 7.389056 ) is 2.0</td>
</tr>
<tr>
<td>log10( x )</td>
<td>logarithm of x (base 10)</td>
<td>log10( 1.0 ) is 0.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>log10( 10.0 ) is 1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>log10( 100.0 ) is 2.0</td>
</tr>
<tr>
<td>fabs( x )</td>
<td>absolute value of x</td>
<td>fabs( 5.0 ) is 5.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>fabs( 0.0 ) is 0.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>fabs( -5.0 ) is 5.0</td>
</tr>
<tr>
<td>ceil( x )</td>
<td>rounds $x$ to the smallest integer not less than $x$</td>
<td>ceil( 9.2 ) is 10.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ceil( -9.8 ) is -9.0</td>
</tr>
<tr>
<td>floor( x )</td>
<td>rounds $x$ to the largest integer not greater than $x$</td>
<td>floor( 9.2 ) is 9.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>floor( -9.8 ) is -10.0</td>
</tr>
<tr>
<td>pow( x, y )</td>
<td>$x$ raised to power $y$ ($x^y$)</td>
<td>pow( 2, 7 ) is 128.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>pow( 9, .5 ) is 3.0</td>
</tr>
<tr>
<td>fmod( x, y )</td>
<td>remainder of $x$ divided by $y$ as a floating point number</td>
<td>fmod( 13.657, 2.333 ) is 1.992</td>
</tr>
<tr>
<td>sin( x )</td>
<td>trigonometric sine of $x$ ($x$ in radians)</td>
<td>sin( 0.0 ) is 0.0</td>
</tr>
<tr>
<td>cos( x )</td>
<td>trigonometric cosine of $x$ ($x$ in radians)</td>
<td>cos( 0.0 ) is 1.0</td>
</tr>
<tr>
<td>tan( x )</td>
<td>trigonometric tangent of $x$ ($x$ in radians)</td>
<td>tan( 0.0 ) is 0.0</td>
</tr>
</tbody>
</table>

Fig. 5.2 Commonly used math library functions.
Functions - Reasons for Use

- Don’t have to repeat the same block of code many times in your code. Make that code block a function and call it when needed.
- Function portability: useful functions can be used in a number of programs.
- Supports the top-down technique for devising a program algorithm. Make an outline and hierarchy of the steps needed to solve your problem and create a function for each step.
- Easy to debug. Get one function working well then move on to the others.
- Easy to modify and expand. Just add more functions to extend program capability.
- For a large programming project, you will code only a small fraction of the program.
- Make program self-documenting and readable.
5.5 Function Definitions

```c
return_type function_name (data type variable name list)
{
    local declarations;
    function statements;
}
```

- where the `return_type` in the function header tells the type of the value returned by the function (default is `int`)
- where the `data type variable name list` tells what arguments the function needs when it is called (and what their types are)
- where `local declarations` in the function body are local constants and variables the function needs for its calculations.
5.5 Function Definitions

Here is an example of a function that calculates \( n! \)

```c
int factorial(int n)
{
    int i, product = 1;
    for (i = 2; i <= n; ++i)
        product *= i;
    return product;
}
```
5.5 Function Definitions

- Function-name: any valid identifier
- Return-value-type: data type of the result (default int)
  - void – indicates that the function returns nothing
- Parameter-list: comma separated list, declares parameters

```c
void write_header(void) {
    printf("Navier-Stokes Equations Solver ");
    printf("v3.45\n");
    printf("Last Modified: ");
    printf("12/04/95 - viscous coefficient added\n");
}
```
5.5 Function Definitions

- A function returns a value to the calling program with the use of the `keyword return`, followed by a data variable or constant value. The return statement can even contain an expression. Some examples
  
  ```
  return 3;
  return n;
  return ++a;
  return (a*b);
  ```

- When a `return` is encountered the following events occur:
  1. execution of the function is terminated and control is passed back to the calling program, and
  2. the function call evaluates to the value of the `return expression`.

- If there is no `return` statement control is passed back when the closing brace of the function is encountered (“falling off the end”).
5.5 Function Definitions

- The data type of the `return expression` must match that of the declared `return_type` for the function.

```c
float add_numbers (float n1, float n2) {
    return n1 + n2; /*legal*/
    return 6;        /*illegal, not the same data type*/
    return 6.0;      /*legal*/
}
```

- It is possible for a function to have multiple `return` statements. For example:

```c
double absolute(double x) {
    if (x>=0.0)
        return x;
    else
        return -x;
}
```
5.5 Using The Functions

- This is the easiest part! To invoke a function, just type its name in your program and be sure to supply arguments (if necessary). A statement using our factorial program would look like

  \[ \text{number} = \text{factorial}(9); \]

- To invoke our write_header function, use this statement

  \[ \text{write_header}(); \]

- When your program encounters a function invocation, control passes to the function. When the function is completed, control passes back to the main program. In addition, if a value was returned, the function call takes on that return value. In the above example, upon return from the \text{factorial} function the statement

  \[ \text{factorial}(9) \rightarrow 362880 \]

- and that integer is assigned to the variable \text{number}. 
Considerations when using Functions

- Some points to keep in mind when calling functions (your own or library’s):
  - The **number** of arguments in the function call **must match** the **number** of arguments in the function definition.
  - The **type** of the arguments in the function call **must match** the **type** of the arguments in the function definition.
  - The **actual arguments** in the function call are **matched up in-order** with the **dummy arguments** in the function definition.
  - The actual arguments are **passed by-value** to the function. The dummy arguments in the function are initialized with the present values of the actual arguments. *Any changes made to the dummy argument in the function will NOT affect the actual argument in the main program.*
Using Function Example

- The independence of actual and dummy arguments is demonstrated in the following program.

```c
#include <stdio.h>
int compute_sum(int n) {
    int sum=0;
    for (;n>0;--n)
        sum+=n;
    printf("Local n in function is %d\n",n);
    return sum; }
main() {
    int n=8,sum;
    printf ("Main n (before call) is %d\n",n);
    sum=compute_sum(n);
    printf ("Main n (after call) is %d\n",n);
    printf ("\nThe sum of integers from 1 to %d is %d\n",n,sum);

Main n (before call) is 8
Local n in function is 0
Main n (after call) is 8
The sum of integers from 1 to 8 is 36
```
5.6 Function Prototypes

- Function prototypes are used to declare a function so that it can be used in a program before the function is actually defined.

```c
#include <stdio.h>
int compute_sum(int n); /* Function Prototype */
main() {
    int n=8,sum;
    printf("Main n (before call) is %d\n",n);
    sum=compute_sum(n);
    printf("Main n (after call) is %d\n",n);
    printf("The sum of integers from 1 to %d is %d\n",n,sum);
}
int compute_sum(int n) {
    int sum=0;
    for(;n>0;--n)
        sum+=n;
    printf("Local n in function is %d\n",n);
    return sum; }
```

Now the program reads in a "natural" order. You know that a function called `compute_sum` will be defined later on, and you see its immediate use in the main program. Perhaps you don’t care about the details of how the sum is computed and you won’t need to read the actual function definition.
/* Fig. 5.3: fig05_03.c 
Creating and using a programmer-defined function */

#include <stdio.h>

int square( int y ); /* function prototype */

/* function main begins program execution */
int main()
{
    int x; /* counter */

    /* loop 10 times and calculate and output square of x each time */
    for ( x = 1; x <= 10; x++ ) {
        printf( "%d ", square( x ) ); /* function call */
    } /* end for */

    printf( "\n" );

    return 0; /* indicates successful termination */
} /* end main */

/* square function definition returns square of an integer */
int square( int y ) /* y is a copy of argument to function */
{
    return y * y; /* returns square of y as an int */
} /* end function square */
fig05_03.c (Part 2 of 2)

Program Output

1  4  9  16  25  36  49  64  81  100
/* Fig. 5.4: fig05_04.c */
#include <stdio.h>

int maximum(int x, int y, int z); /* function prototype */

/* function main begins program execution */
int main()
{
    int number1; /* first integer */
    int number2; /* second integer */
    int number3; /* third integer */

    printf("Enter three integers: ");
    scanf("%d%d%d", &number1, &number2, &number3);

    /* number1, number2 and number3 are arguments to the maximum function call */
    printf("Maximum is: %d\n", maximum(number1, number2, number3));

    return 0; /* indicates successful termination */
} /* end main */
/* Function maximum definition */
/* x, y and z are parameters */
int maximum( int x, int y, int z )
{
    int max = x; /* assume x is largest */
    if ( y > max ) /* if y is larger than max, assign y to max */
        max = y;
    } /* end if */
    if ( z > max ) /* if z is larger than max, assign z to max */
        max = z;
    } /* end if */
    return max; /* max is largest value */
} /* end function maximum */

Program Output
Enter three integers: 22 85 17
Maximum is: 85
Enter three integers: 85 22 17
Maximum is: 85
Enter three integers: 22 17 85
Maximum is: 85
5.6 Function Prototypes

Promotion rules and conversions
Converting to lower types can lead to errors

<table>
<thead>
<tr>
<th>Data types</th>
<th>printf conversion specifications</th>
<th>scanf conversion specifications</th>
</tr>
</thead>
<tbody>
<tr>
<td>long double</td>
<td>%Lf</td>
<td>%Lf</td>
</tr>
<tr>
<td>double</td>
<td>%f</td>
<td>%lf</td>
</tr>
<tr>
<td>float</td>
<td>%f</td>
<td>%f</td>
</tr>
<tr>
<td>unsigned long int</td>
<td>%lu</td>
<td>%lu</td>
</tr>
<tr>
<td>long int</td>
<td>%ld</td>
<td>%ld</td>
</tr>
<tr>
<td>unsigned int</td>
<td>%u</td>
<td>%u</td>
</tr>
<tr>
<td>int</td>
<td>%d</td>
<td>%d</td>
</tr>
<tr>
<td>short</td>
<td>%hd</td>
<td>%hd</td>
</tr>
<tr>
<td>char</td>
<td>%c</td>
<td>%c</td>
</tr>
</tbody>
</table>

Fig. 5.5 Promotion hierarchy for data types.
5.7 Header Files

• Header files
  – Contain function prototypes for library functions
    – `<stdlib.h>`, `<math.h>`, etc
  – Load with `#include <filename>`
    `#include <math.h>`

• Custom header files
  – Create file with functions
  – Save as `filename.h`
  – Load in other files with `#include "filename.h"`
  – Reuse functions
# 5.7 Header Files

<table>
<thead>
<tr>
<th>Standard library header</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;assert.h&gt;</code></td>
<td>Contains macros and information for adding diagnostics that aid program debugging.</td>
</tr>
<tr>
<td><code>&lt;ctype.h&gt;</code></td>
<td>Contains function prototypes for functions that test characters for certain properties, and function prototypes for functions that can be used to convert lowercase letters to uppercase letters and vice versa.</td>
</tr>
<tr>
<td><code>&lt;errno.h&gt;</code></td>
<td>Defines macros that are useful for reporting error conditions.</td>
</tr>
<tr>
<td><code>&lt;float.h&gt;</code></td>
<td>Contains the floating point size limits of the system.</td>
</tr>
<tr>
<td><code>&lt;limits.h&gt;</code></td>
<td>Contains the integral size limits of the system.</td>
</tr>
<tr>
<td><code>&lt;locale.h&gt;</code></td>
<td>Contains function prototypes and other information that enables a program to be modified for the current locale on which it is running. The notion of locale enables the computer system to handle different conventions for expressing data like dates, times, dollar amounts and large numbers throughout the world.</td>
</tr>
<tr>
<td><code>&lt;math.h&gt;</code></td>
<td>Contains function prototypes for math library functions.</td>
</tr>
<tr>
<td><code>&lt;setjmp.h&gt;</code></td>
<td>Contains function prototypes for functions that allow bypassing of the usual function call and return sequence.</td>
</tr>
<tr>
<td><code>&lt;signal.h&gt;</code></td>
<td>Contains function prototypes and macros to handle various conditions that may arise during program execution.</td>
</tr>
<tr>
<td><code>&lt;stdarg.h&gt;</code></td>
<td>Defines macros for dealing with a list of arguments to a function whose number and types are unknown.</td>
</tr>
<tr>
<td><code>&lt;stddef.h&gt;</code></td>
<td>Contains common definitions of types used by C for performing certain calculations.</td>
</tr>
<tr>
<td><code>&lt;stdio.h&gt;</code></td>
<td>Contains function prototypes for the standard input/output library functions, and information used by them.</td>
</tr>
<tr>
<td><code>&lt;stdlib.h&gt;</code></td>
<td>Contains function prototypes for conversions of numbers to text and text to numbers, memory allocation, random numbers, and other utility functions.</td>
</tr>
<tr>
<td><code>&lt;string.h&gt;</code></td>
<td>Contains function prototypes for string processing functions.</td>
</tr>
<tr>
<td><code>&lt;time.h&gt;</code></td>
<td>Contains function prototypes and types for manipulating the time and date.</td>
</tr>
</tbody>
</table>

**Fig. 5.6** Some of the standard library header.
5.11 Storage Classes

Every variable in C actually has two attributes: its data type and its storage class. The storage class refers to the manner in which memory is allocated for the variable. The storage class also determines the scope of the variable, that is, what parts of a program the variable’s name has meaning. In C, the four possible Storage classes are

- auto
- extern
- static
- register
5.11 Storage Classes – Auto

- This is the **default classification** for all variables declared within a function body [including `main()`].
- Automatic variables are truly **local**.
- They exist and their names have meaning only while the function is being executed.
- They are unknown to other functions.
- When the function is exited, the values of automatic variables are not retained.
- They are recreated each time the function is called.

```c
auto double x, y;
```
5.11 Storage Classes – Extern

- In contrast to auto, extern variables are **global**.
- If a variable is declared at the beginning of a program outside all functions [including `main()`] it is classified as an external by default.
- **External variables can be accessed and changed by any function in the program.**
- Their storage is in permanent memory, and thus never disappear or need to be recreated.

What is the advantage of using global variables?

**It is a method of transmitting information between functions in a program without using arguments.**
There are **two disadvantages** of global variables versus arguments. **First**, the function is much less portable to other programs. **Second**, is the concept of **local dominance**. If a local variable has the **same name** as a global variable, only the local variable is changed while in the function. Once the function is exited, the global variable has the same value as when the function started.
5.11 Storage Classes – Static

• A static variable is a local variable that retains its latest value when a function is recalled.
  – Keep value after function ends
  – Only known in their own function
• Static variables are created and initialized once on the first call to the function.
• With clever programming, one can use static variables to enable a function to do different things depending on how many times it has been called. (Consider a function that counts the number of times it has been called).
5.11 Storage Classes – Register

• It is often true that the time bottleneck in computer calculations is the time it takes to fetch a variable from memory and store its value in a register where the CPU can perform some calculation with it.
• So for performance reasons, it is sometimes advantageous to store variables directly in registers. This strategy is most often used with loop counter variables, as shown below.

```c
register int i;
for (i=0; i<n; ++i)
...
```

Note that this type can only be used for automatic variables.
5.12 Scope Rules

• File scope
  – Identifier defined outside function, known in all functions
  – Used for global variables, function definitions, function prototypes

• Function scope
  – Can only be referenced inside a function body
  – Used only for labels (start:, case:, etc.)
5.12 Scope Rules

• Block scope
  – Identifier declared inside a block
    • Block scope begins at definition, ends at right brace
  – Used for variables, function parameters (local variables of function)
  – Outer blocks "hidden" from inner blocks if there is a variable with the same name in the inner block

• Function prototype scope
  – Used for identifiers in parameter list
/* Fig. 5.12: fig05_12.c 
A scoping example */

#include <stdio.h>

void useLocal( void ); /* function prototype */
void useStaticLocal( void ); /* function prototype */
void useGlobal( void ); /* function prototype */

int x = 1; /* global variable */

/* function main begins program execution */
int main()
{
    int x = 5; /* local variable to main */

    printf("local x in outer scope of main is %d\n", x);

    { /* start new scope */
        int x = 7; /* local variable to new scope */

        printf("local x in inner scope of main is %d\n", x);
    } /* end new scope */

    printf("local x in outer scope of main is %d\n", x);
} /* end function main() */
```c
useLocal(); /* useLocal has automatic local x */
useStaticLocal(); /* useStaticLocal has static local x */
useGlobal(); /* useGlobal uses global x */
useLocal(); /* useLocal reinitializes automatic local x */
useStaticLocal(); /* static local x retains its prior value */
useGlobal(); /* global x also retains its value */

printf( "local x in main is %d\n", x );

return 0; /* indicates successful termination */

} /* end main */

/* useLocal reinitializes local variable x during each call */
void useLocal( void )
{
    int x = 25; /* initialized each time useLocal is called */

    printf( "\nlocal x in a is %d after entering a\n", x );
    x++;
    printf( "local x in a is %d before exiting a\n", x );
} /* end function useLocal */
```
/* useStaticLocal initializes static local variable x only the first time the function is called; value of x is saved between calls to this function */

void useStaticLocal( void )
{
    /* initialized only first time useStaticLocal is called */
    static int x = 50;

    printf( "\nlocal static x is %d on entering b\n", x );
    x++;
    printf( "local static x is %d on exiting b\n", x );
} /* end function useStaticLocal */

/* function useGlobal modifies global variable x during each call */
void useGlobal( void )
{
    printf( "\nglobal x is %d on entering c\n", x );
    x *= 10;
    printf( "global x is %d on exiting c\n", x );
} /* end function useGlobal */
local x in outer scope of main is 5
local x in inner scope of main is 7
local x in outer scope of main is 5

local x in a is 25 after entering a
local x in a is 26 before exiting a

local static x is 50 on entering b
local static x is 51 on exiting b

global x is 1 on entering c
global x is 10 on exiting c

local x in a is 25 after entering a
local x in a is 26 before exiting a

local static x is 51 on entering b
local static x is 52 on exiting b

global x is 10 on entering c
global x is 100 on exiting c
local x in main is 5
5.13 Recursion

• Recursive functions
  – Functions that call themselves
  – Can only solve a base case
  – Divide a problem up into
    • What it can do
    • What it cannot do
      – What it cannot do resembles original problem
      – The function launches a new copy of itself (recursion step) to solve what it cannot do
  – Eventually base case gets solved
    • Gets plugged in, works its way up and solves whole problem
5.13 Recursion

- **Example: factorials**
  - $5! = 5 \times 4 \times 3 \times 2 \times 1$
  - Notice that
    - $5! = 5 \times 4!$
    - $4! = 4 \times 3! \ldots$
  - Can compute factorials recursively
  - Solve base case ($1! = 0! = 1$) then plug in
    - $2! = 2 \times 1! = 2 \times 1 = 2$;
    - $3! = 3 \times 2! = 3 \times 2 = 6$;
5.13 Recursion

(a) Sequence of recursive calls
(b) Values returned from each recursive call.

5! = 5 \times 4!

5 \times 4!

4! = 4 \times 3!

4 \times 3!

3! = 3 \times 2!

3 \times 2!

2! = 2 \times 1!

2 \times 1!

1 returned

Final value = 120

5! = 5 \times 4! = 5 \times 24 = 120 is returned

4! = 4 \times 3! = 4 \times 6 = 24 is returned

3! = 3 \times 2! = 3 \times 2 = 6 is returned

2! = 2 \times 1! = 2 \times 1 = 2 is returned

1 returned
/* Fig. 5.14: fig05_14.c
Recursive factorial function */
#include <stdio.h>

long factorial( long number ); /* function prototype */

/* function main begins program execution */
int main()
{
    int i; /* counter */

    /* loop 10 times. During each iteration, calculate factorial( i ) and display result */
    for ( i = 1; i <= 10; i++ ) {
        printf( "%2d! = %ld\n", i, factorial( i ) );
    } /* end for */

    return 0; /* indicates successful termination */
} /* end main */
```c
22   /* recursive definition of function factorial */
23   long factorial( long number )
24   {
25       /* base case */
26       if ( number <= 1 ) {
27           return 1;
28       } /* end if */
29   } /* end else */
30   return ( number * factorial( number - 1 ) );
31   } /* end else */
32   } /* end function factorial */
```

```
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
```
5.14 Example Using Recursion: The Fibonacci Series

- Fibonacci series: 0, 1, 1, 2, 3, 5, 8...
  - Each number is the sum of the previous two
  - Can be solved recursively:
    - \( \text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2) \)
  - Code for the \text{fibonacci} function
    ```c
    long fibonacci( long n )
    {
      if (n == 0 || n == 1)  // base case
        return n;
      else
        return fibonacci( n - 1) + fibonacci( n - 2 );
    }
    ```
Example Using Recursion: The Fibonacci Series

```
5.14 Example Using Recursion: The Fibonacci Series

return fibonacci(3)

fibonacci(3) = fibonacci(2) + fibonacci(1)

return fibonacci(2)
 return fibonacci(1) + fibonacci(0)

return fibonacci(1) = fibonacci(0) + fibonacci(-1)
 return fibonacci(0) = 1

return fibonacci(0) = 0

```

Diagram:

```
      fibonacci(3)
       /   \
      /     \fibonacci(2) + fibonacci(1)
    /       \
   /         \fibonacci(1) + fibonacci(0)
  /           \
 return 1
```

```
return 1
```
/* Fig. 5.15: fig05_15.c 
   Recursive fibonacci function */

#include <stdio.h>

long fibonacci( long n ); /* function prototype */

/* function main begins program execution */
int main()
{
    long result; /* fibonacci value */
    long number; /* number input by user */

    /* obtain integer from user */
    printf( "Enter an integer: " );
    scanf( "%ld", &number );

    /* calculate fibonacci value for number input by user */
    result = fibonacci( number );

    /* display result */
    printf( "Fibonacci( %ld ) = %ld\n", number, result );

    return 0; /* indicates successful termination */
}
/* end main */
/** Recursive definition of function fibonacci */
long fibonacci( long n )
{
    /* base case */
    if ( n == 0 || n == 1 ) {
        return n;
    } /* end if */
    /* recursive step */
    else {
        return fibonacci( n - 1 ) + fibonacci( n - 2 );
    } /* end else */
} /* end function fibonacci */

Enter an integer: 0
Fibonacci( 0 ) = 0

Enter an integer: 1
Fibonacci( 1 ) = 1

Enter an integer: 2
Fibonacci( 2 ) = 1

Enter an integer: 3
Fibonacci( 3 ) = 2

Enter an integer: 4
Fibonacci( 4 ) = 3
Enter an integer: 5
Fibonacci( 5 ) = 5

Enter an integer: 6
Fibonacci( 6 ) = 8

Enter an integer: 10
Fibonacci( 10 ) = 55

Enter an integer: 20
Fibonacci( 20 ) = 6765

Enter an integer: 30
Fibonacci( 30 ) = 832040

Enter an integer: 35
Fibonacci( 35 ) = 9227465
5.15 Recursion vs. Iteration

- Repetition
  - Iteration: explicit loop
  - Recursion: repeated function calls

- Termination
  - Iteration: loop condition fails
  - Recursion: base case recognized

- Both can have infinite loops

- Balance
  - Choice between performance (iteration) and good software engineering (recursion)
5.9 Random Number Generation

- **rand function**
  - Load `<stdlib.h>`
  - Returns "random" number between 0 and `RAND_MAX` (at least 32767)
    ```c
    i = rand();
    ```
  - Pseudorandom
    - Preset sequence of "random" numbers
    - Same sequence for every function call

- **Scaling**
  - To get a random number between 1 and n
    ```c
    1 + ( rand() % n )
    ```
    - `rand() % n` returns a number between 0 and n - 1
    - Add 1 to make random number between 1 and n
    ```c
    1 + ( rand() % 6)
    ```
    - number between 1 and 6
5.9 Random Number Generation

- **srand function**
  - `<stdlib.h>`
  - Takes an integer seed and jumps to that location in its "random" sequence
    ```c
    srand( seed );
    ```
  - `srand( time( NULL ) );` /*load `<time.h>` */

- **time( NULL )**
  - Returns the time at which the program was compiled in seconds
  - "Randomizes" the seed
/* Fig. 5.7: fig05_07.c */
Shifted, scaled integers produced by \texttt{1 + \texttt{rand()} \% 6} */

#include <stdio.h>
#include <stdlib.h>

/* function main begins program execution */
int main()
{
  int i; /* counter */

  /* loop 20 times */
  for ( i = 1; i <= 20; i++ ) {

    /* pick random number from 1 to 6 and output it */
    printf( "%10d", 1 + ( \texttt{rand()} \% 6 ) );

    /* if counter is divisible by 5, begin new line of output */
    if ( i \% 5 == 0 ) {
      printf( "\n" );
    } /* end if */

  } /* end for */

  return 0; /* indicates successful termination */

} /* end main */
<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>2</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>
/* Fig. 5.8: fig05_08.c
   Roll a six-sided die 6000 times */
#include <stdio.h>
#include <stdlib.h>

/* function main begins program execution */
int main()
{
    int frequency1 = 0; /* rolled 1 counter */
    int frequency2 = 0; /* rolled 2 counter */
    int frequency3 = 0; /* rolled 3 counter */
    int frequency4 = 0; /* rolled 4 counter */
    int frequency5 = 0; /* rolled 5 counter */
    int frequency6 = 0; /* rolled 6 counter */

    int roll; /* roll counter */
    int face; /* represents one roll of the die, value 1 to 6 */

    /* loop 6000 times and summarize results */
    for (roll = 1; roll <= 6000; roll++) {
        face = 1 + rand() % 6; /* random number from 1 to 6 */
/* determine face value and increment appropriate counter */
switch ( face ) {
    case 1:       /* rolled 1 */
        ++frequency1;
        break;
    case 2:       /* rolled 2 */
        ++frequency2;
        break;
    case 3:       /* rolled 3 */
        ++frequency3;
        break;
    case 4:       /* rolled 4 */
        ++frequency4;
        break;
    case 5:       /* rolled 5 */
        ++frequency5;
        break;
}
```c

    case 6: /* rolled 6 */
        ++frequency6;
        break;
    } /* end switch */

    } /* end for */

    /* display results in tabular format */
    printf( "%s%13s\n", "Face", "Frequency" );
    printf( "   1%13d\n", frequency1 );
    printf( "   2%13d\n", frequency2 );
    printf( "   3%13d\n", frequency3 );
    printf( "   4%13d\n", frequency4 );
    printf( "   5%13d\n", frequency5 );
    printf( "   6%13d\n", frequency6 );

    return 0; /* indicates successful termination */

} /* end main */

```

Program Output

<table>
<thead>
<tr>
<th>Face</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1003</td>
</tr>
<tr>
<td>2</td>
<td>1017</td>
</tr>
<tr>
<td>3</td>
<td>983</td>
</tr>
<tr>
<td>4</td>
<td>994</td>
</tr>
<tr>
<td>5</td>
<td>1004</td>
</tr>
<tr>
<td>6</td>
<td>999</td>
</tr>
</tbody>
</table>
/* Fig. 5.9: fig05_09.c */
#include <stdlib.h>
#include <stdio.h>

/* function main begins program execution */
int main()
{
  int i; /* counter */
  unsigned seed; /* number used to seed random number generator */

  printf( "Enter seed: " );
  scanf( "%u", &seed );

  srand( seed ); /* seed random number generator */

  /* loop 10 times */
  for ( i = 1; i <= 10; i++ ) {
    /* pick a random number from 1 to 6 and output it */
    printf( "%10d", 1 + ( rand() % 6 ) );
  }
}
/* if counter is divisible by 5, begin a new line of output */
if ( i % 5 == 0 ) {
    printf( "\n" );
} /* end if */

} /* end for */

return 0; /* indicates successful termination */

} /* end main */
5.10 Example: A Game of Chance

- Craps simulator
- Rules
  - Roll two dice
    - 7 or 11 on first throw, player wins
    - 2, 3, or 12 on first throw, player loses
    - 4, 5, 6, 8, 9, 10 - value becomes player's "point"
  - Player must roll his point before rolling 7 to win
/* Fig. 5.10: fig05_10.c 
   Craps */

#include <stdio.h>
#include <stdlib.h>
#include <time.h> /* contains prototype for function time */

/* enumeration constants represent game status */
enum Status { CONTINUE, WON, LOST };

int rollDice( void ); /* function prototype */

/* function main begins program execution */
int main()
{
    int sum; /* sum of rolled dice */
    int myPoint; /* point earned */

    enum Status gameStatus; /* can contain CONTINUE, WON, or LOST */

    /* randomize random number generator using current time */
    srand( time( NULL ) );

    sum = rollDice( ); /* first roll of the dice */
/* determine game status based on sum of dice */
switch( sum ) {

    /* win on first roll */
    case 7:
    case 11:
        gameStatus = WON;
        break;

    /* lose on first roll */
    case 2:
    case 3:
    case 12:
        gameStatus = LOST;
        break;

    /* remember point */
    default:
        gameStatus = CONTINUE;
        myPoint = sum;
        printf( "Point is %d\n", myPoint );
        break; /* optional */
}

} /* end switch */

/* while game not complete */
while ( gameStatus == CONTINUE ) {
    sum = rollDice(  ); /* roll dice again */

    /* determine game status */
    if ( sum == myPoint ) { /* win by making point */
        gameStatus = WON;
    } /* end if */
    else {
        if ( sum == 7 ) { /* lose by rolling 7 */
            gameStatus = LOST;
        } /* end if */
    } /* end else */

} /* end while */

/* display won or lost message */
if ( gameStatus == WON ) {
    printf( "Player wins\n" );
} /* end if */
else {
    printf( "Player loses\n" );
} /* end else */
```c
return 0; /* indicates successful termination */
}

/* roll dice, calculate sum and display results */
int rollDice(void)
{
    int die1;    /* first die */
    int die2;    /* second die */
    int workSum; /* sum of dice */

    die1 = 1 + ( rand() % 6 ); /* pick random die1 value */
    die2 = 1 + ( rand() % 6 ); /* pick random die2 value */
    workSum = die1 + die2;    /* sum die1 and die2 */

    /* display results of this roll */
    printf("Player rolled %d + %d = %d\n", die1, die2, workSum);
    
    return workSum; /* return sum of dice */
}
```
Player rolled 5 + 6 = 11
Player wins

Player rolled 4 + 1 = 5
Point is 5
Player rolled 6 + 2 = 8
Player rolled 2 + 1 = 3
Player rolled 3 + 2 = 5
Player wins

Player rolled 1 + 1 = 2
Player loses

Player rolled 1 + 4 = 5
Point is 5
Player rolled 3 + 4 = 7
Player loses