Chapter 5 Part 2 C Functions

C How to Program

Random Number Generation

- The element of chance can be introduced into computer applications by using the C standard library function `rand` from the `<stdlib.h>` header.
- Consider the following statement:

```
i = rand();
```

- The `rand` function generates an integer between 0 and `RAND_MAX` (a symbolic constant defined in the `<stdlib.h>` header).

Random Number Generation (Cont.)

- Standard C states that the value of `RAND_MAX` must be at least 32767.
- If `rand` produces integers at random, every number between 0 and `RAND_MAX` has an equal chance (or probability) of being chosen each time `rand` is called.
- The range of values produced directly by `rand` is often different from what's needed in a specific application.

Rolling a Six-Sided Die

- To demonstrate `rand`, let's develop a program to simulate 20 rolls of a six-sided die and print the value of each roll.
- The function prototype for function `rand` is in `<stdlib.h>`.
- We use the remainder operator (`%`) in conjunction with `rand` as follows

```
rand() % 6
```

- to produce integers in the range 0 to 5.

Rolling a Six-Sided Die (Cont.)

- This is called scaling
- The number 6 is called the scaling factor.

- We then shift the range of numbers produced by adding 1 to our previous result.

- The output of Fig. 5.7 confirms that the results are in the range 1 to 6—the actual random values chosen might vary by compiler.

```c
// Fig. 5.11: fig05_11.c
// Shifted, scaled random integers produced by 1 + rand() % 6.
#include <stdio.h>
#include <stdlib.h>

// function main begins program execution
int main( void )
{
   unsigned 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 + ( rand() % 6 ) );

      // if counter is divisible by 5, begin new line of output
      if ( i % 5 == 0 ) {
         puts( "" );
      } // end if
   } // end for
} // end main
```

```
         6         6         5         5         6
         5         1         1         5         3
         6         6         2         4         2
         6         2         3         4         1
```

Fig. 5.11 | Shifted, scaled random integers produced by 1 + rand() % 6. (Part 2 of 2.)

Randomizing the Random Number Generator

- Executing the program again produces exactly the same sequence of values.

- How can these be *random* numbers? Ironically, this repeatability is an important characteristic of function `rand`.

- When *debugging* a program, this repeatability is essential for proving that corrections to a program work properly.

Randomizing the Random Number Generator (Cont.)

- So, the function `rand` actually generates pseudorandom numbers.
- Calling `rand` repeatedly produces a sequence of numbers that appears to be random.
- However, the sequence repeats itself each time the program is executed.
- Once a program has been thoroughly debugged, it can be conditioned to produce a different sequence of random numbers for each execution.

Randomizing the Random Number Generator (Cont.)

- This is called randomizing and is accomplished with the standard library function `srand`.
- Function `srand` takes an `unsigned` integer argument and seeds function `rand` to produce a different sequence of random numbers for each execution of the program.

Randomizing the Random Number Generator (Cont.)

- Function `srand` takes an `unsigned int` value as an argument.
- The conversion specifier `%u` is used to read an `unsigned int` value with `scanf`.
- The function prototype for `srand` is found in `<stdlib.h>`.

```c
// Fig. 5.13: fig05_13.c
// Randomizing the die-rolling program.
#include <stdlib.h>
#include <stdio.h>

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

   printf( "%s", "Enter seed: " );
   scanf( "%u", &seed ); // note %u for unsigned int

   srand( seed );   // seed the 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 ) {
         puts( "" );
      } // end if
   } // end for
} // end main
```

```
Enter seed: 67
         6         1         4         6         2
         1         6         1         6         4
```

```
Enter seed: 867
         2         4         6         1         6
         1         1         3         6         2
```

```
Enter seed: 67
         6         1         4         6         2
         1         6         1         6         4
```

Randomizing the Random Number Generator (Cont.)

- Notice that a different sequence of random numbers is obtained each time the program is run, provided that a different seed is supplied.

- To randomize without entering a seed each time, use a statement like

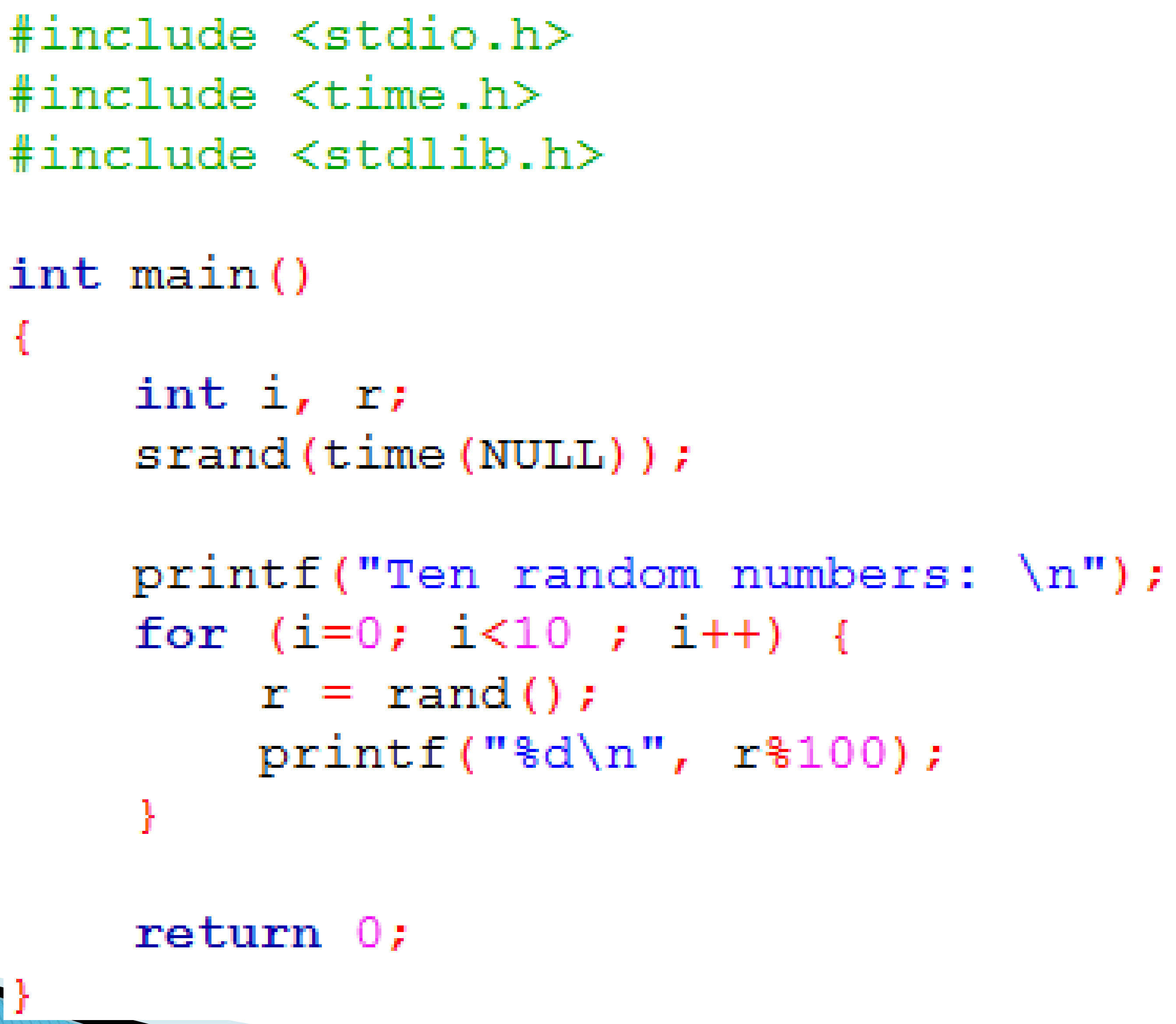
```
  srand( time( NULL ) );
```

- This causes the computer to read its clock to obtain the value for the seed automatically.

Randomizing the Random Number Generator (Cont.)

- The function `time` returns the number of seconds that have passed since midnight on January 1, 1970.
- This value is converted to an unsigned integer and used as the seed to the random number generator.
- The function prototype for `time` is in <`time.h`>.

Type this program in codeblocks



```c
#include <stdio.h>
#include <time.h>
#include <stdlib.h>

int main()
{
    int i, r;
    srand(time(NULL));

    printf("Ten random numbers: \n");
    for (i=0; i<10 ; i++) {
        r = rand();
        printf("%d\n", r%100);
    }

    return 0;
}
```

Storage Classes

- The basic attributes of variables include name, type, size and value.
- Each identifier in a program has other attributes, including storage class, storage duration, scope and linkage.
- An identifier’s storage class determines its storage duration, scope and linkage.

Storage Classes (Cont.)

- An identifier’s storage duration is the period during which the identifier exists *in memory*.
- Some exist briefly, some are repeatedly created and destroyed, and others exist for the program’s entire execution.

- An identifier’s scope is where the identifier can be referenced in a program.
- Some can be referenced throughout a program, others from only portions of a program.

Storage Classes (Cont.)

- The storage-class specifiers can be split into automatic storage duration and static storage duration.
- Keyword `auto` can be used to declare variables of automatic storage duration.
- Variables with automatic storage duration are created when the block in which they're defined is entered; they exist while the block is active, and they're destroyed when the block is exited.

Automatic Storage Duration

Local Variables

- Only variables can have automatic storage duration.
- A function's local variables (those declared in the parameter list or function body) have automatic storage duration.
- Local variables have automatic storage duration by *default*, so keyword `auto` is rarely used.

Static Storage Duration

- The keyword `static` is used in the declarations of identifiers for variables of static storage duration.

- Identifiers of static storage duration exist from the time at which the program begins execution until the program terminates.

- For static variables, storage is allocated and initialized *only once*, *before* the program begins execution.

Static Storage Duration (Cont.)

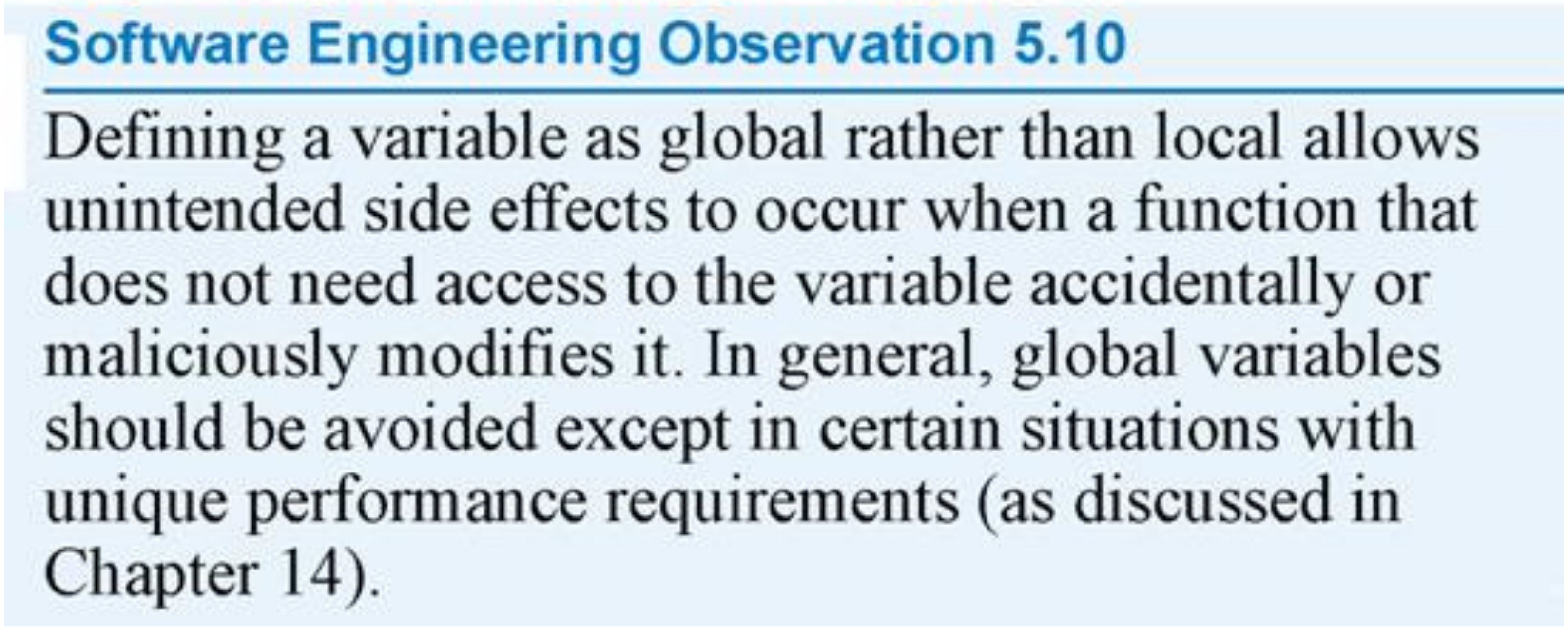
- However, even though the variables names exist from the start of program execution, this does not mean that these identifiers can be accessed throughout the program.
- There are several types of identifiers with static storage duration: global variables are by default static, and local variables that are declared with the storage-class specifier `static`.

Static Storage Duration (Cont.)

- Local variables declared with the keyword `static` are still known only in the function in which they're defined, but unlike automatic variables, `static` local variables retain their value when the function is exited.
- The next time the function is called, the `static` local variable contains the value it had when the function last exited.
- The following statement declares local variable `count` to be `static` and initializes it to 1.
  - `static int count = 1;`

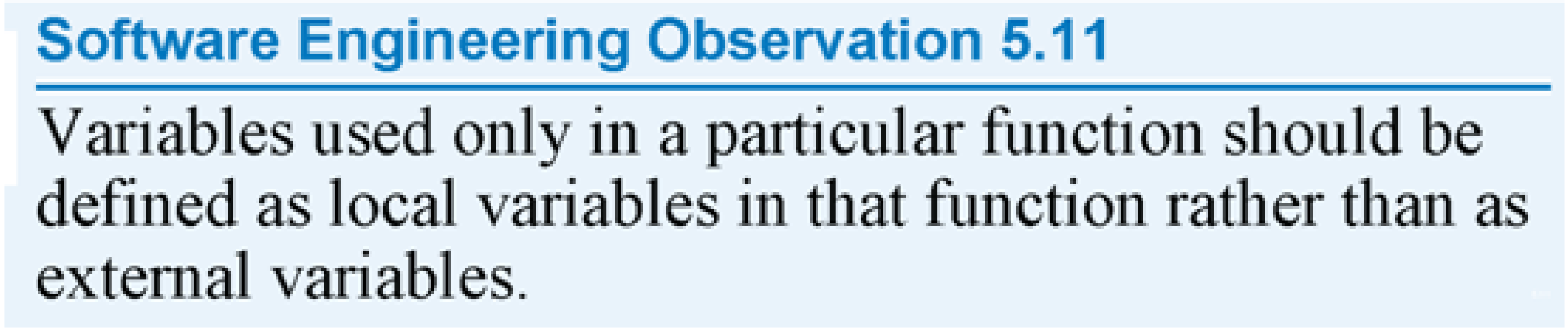
Global Variables

- Global variables are created by placing variable declarations *outside* any function definition, and they retain their values throughout the execution of the program.
- Global variables and functions can be referenced by any function that follows their declarations or definitions in the file.



Software Engineering Observation 5.10

Defining a variable as global rather than local allows unintended side effects to occur when a function that does not need access to the variable accidentally or maliciously modifies it. In general, global variables should be avoided except in certain situations with unique performance requirements (as discussed in Chapter 14).



Software Engineering Observation 5.11

Variables used only in a particular function should be defined as local variables in that function rather than as external variables.

Scope Rules

- The scope of an identifier is the portion of the program in which the identifier can be referenced.
- For example, when we define a local variable in a block, it can be referenced only following its definition in that block.
- The four identifier scopes are file scope, function scope, block scope, and function-prototype scope.

Scope Rules – file scope

- An identifier declared outside any function has file scope.
- Such an identifier is “known” (i.e., accessible) in all functions from the point at which the identifier is declared until the end of the file.
- Global variables, function definitions, and function prototypes placed outside a function all have file scope.

Scope Rules – function scope

- Labels (identifiers followed by a colon such as `start:`)
- Labels can be used anywhere in the function in which they appear, but cannot be referenced outside the function body.
- Labels are used in `switch` statements (as `case` labels) and in `goto` statements (see Chapter 14).

Scope Rules – block scope

- Identifiers defined inside a block have block scope.
- Block scope begins with the left brace (`{`)
- ends at the terminating right brace (`}`) of the block.

- Local variables defined at the beginning of a function have block scope as do function parameters, which are considered local variables by the function.

- Any block may contain variable definitions.

Scope Rules – function prototype scope

- The only identifiers with function-prototype scope are those used in the parameter list of a function prototype.
- Function prototypes do not require names in the parameter list—only types are required.
- If a name is used in the parameter list of a function prototype, the compiler ignores the name.
- Identifiers used in a function prototype can be reused elsewhere in the program without ambiguity.

Scope Rules (Cont.)

- Figure 5.16 demonstrates scoping issues with global variables, automatic local variables, and `static` local variables.

Type this program in Codeblocks – part 1

```c
// Fig. 5.16: fig05_16.c
// Scoping.
#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( void )
{
   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 );

   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( "\nlocal x in main is %d\n", x );
} // end main
```

Type this program in Codeblocks – part 2

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

   printf( "\nlocal x in useLocal is %d after entering useLocal\n", x );
   ++x;
   printf( "local x in useLocal is %d before exiting useLocal\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 once before program startup
   static int x = 50;

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

// function useGlobal modifies global variable x during each call
void useGlobal( void )
{
   printf( "\nglobal x is %d on entering useGlobal\n", x );
   x *= 10;
   printf( "global x is %d on exiting useGlobal\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 useLocal is 25 after entering useLocal
local x in useLocal is 26 before exiting useLocal

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

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

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

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

global x is 10 on entering useGlobal
global x is 100 on exiting useGlobal

local x in main is 5
```

Fig. 5.16 | Scoping. (Part 4 of 4.)

Recursion

- The programs we’ve discussed are generally structured as functions that call one another in a disciplined, hierarchical manner.
- For some types of problems, it’s useful to have functions call themselves.
- A recursive function is a function that calls itself either directly or indirectly through another function.

Recursion (Cont.)

- A recursive function is called to solve a problem.
- The function actually knows how to solve only the simplest case(s), or so-called base case(s).
- Note:
- Recursion is a *complex topic* discussed at length in upper-level computer science courses.
- In this course, only simple examples of recursion are presented.

Recursion (Cont.)

- If the function is called with a base case, the function simply returns a result.
- If the function is called with a more complex problem, the function divides the problem into two conceptual pieces: a piece that the function knows how to do and a piece that it does not know how to do.
- To make recursion feasible, the latter piece must resemble the original problem, but be a slightly simpler or smaller version.

Recursion (Cont.)

- Because this new problem looks like the original problem, the function launches (calls) a fresh copy of itself to go to work on the smaller problem—this is referred to as a recursive call or the recursion step.
- The recursion step also includes the keyword `return`, because its result will be combined with the portion of the problem the function knew how to solve to form a result that will be passed back to the original caller.
- The recursion step executes while the original call to the function has not yet finished executing.

Recursion (Cont.)

- The recursion step can result in many more such recursive calls, as the function keeps dividing each problem it’s called with into two conceptual pieces.

- For the recursion to terminate, each time the function calls itself with a slightly simpler version of the original problem, this sequence of smaller problems must eventually converge on the base case.

- When the function recognizes the base case, returns a result to the previous copy of the function, and a sequence of returns ensues all the way up the line until the original call of the function eventually returns the final result to `main`.

Recursively Calculating Factorials

- The factorial of a nonnegative integer *n*, written *n!* (pronounced "*n* factorial"), is the product
  - $n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$

  with 1! equal to 1, and 0! defined to be 1.

- For example, 5! is the product 5 * 4 * 3 * 2 * 1, which is equal to 120.

- The factorial of an integer, `number`, greater than or equal to `0` can be calculated iteratively (nonrecursively) using a `for` statement as follows:

```
factorial = 1;
for ( counter = number; counter >= 1; counter-- )
   factorial *= counter;
```

Recursively Calculating Factorials

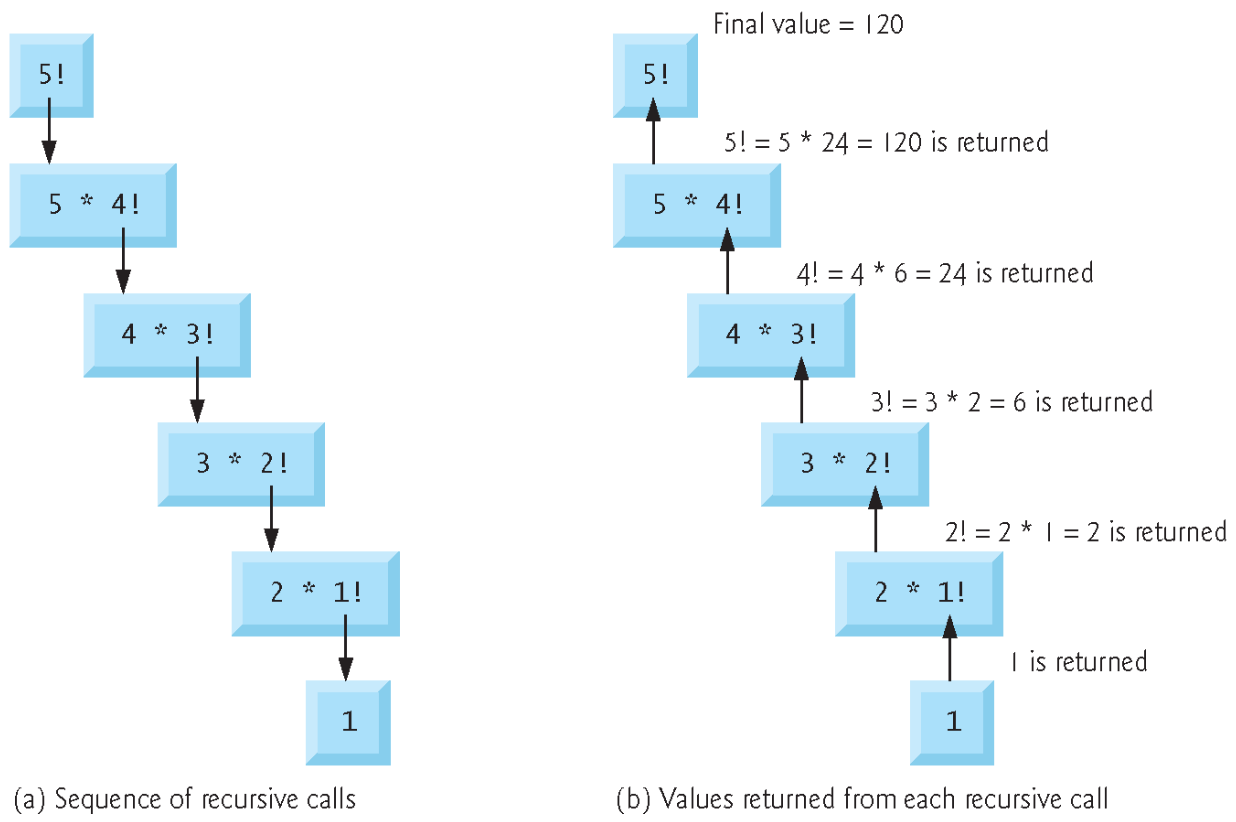
- A *recursive* definition of the factorial function is arrived at by observing the following relationship:

  $$n! = n \cdot (n-1)!$$

- For example, 5! is clearly equal to 5 * 4! as is shown by the following:

  $$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$$
  $$5! = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1)$$
  $$5! = 5 \cdot (4!)$$

- The evaluation of 5! would proceed as shown in Fig. 5.17.



5!
5 * 4!
4 * 3!
3 * 2!
2 * 1!
1
(a) Sequence of recursive calls
Final value = 120
5!
5! = 5 * 24 = 120 is returned
5 * 4!
4! = 4 * 6 = 24 is returned
4 * 3!
3! = 3 * 2 = 6 is returned
3 * 2!
2! = 2 * 1 = 2 is returned
2 * 1!
1 is returned
1
(b) Values returned from each recursive call

Fig. 5.17 | Recursive evaluation of 5!.

Recursion (Cont.)

- If `number` is indeed less than or equal to `1`, `factorial` returns `1`, no further recursion is necessary, and the program terminates.

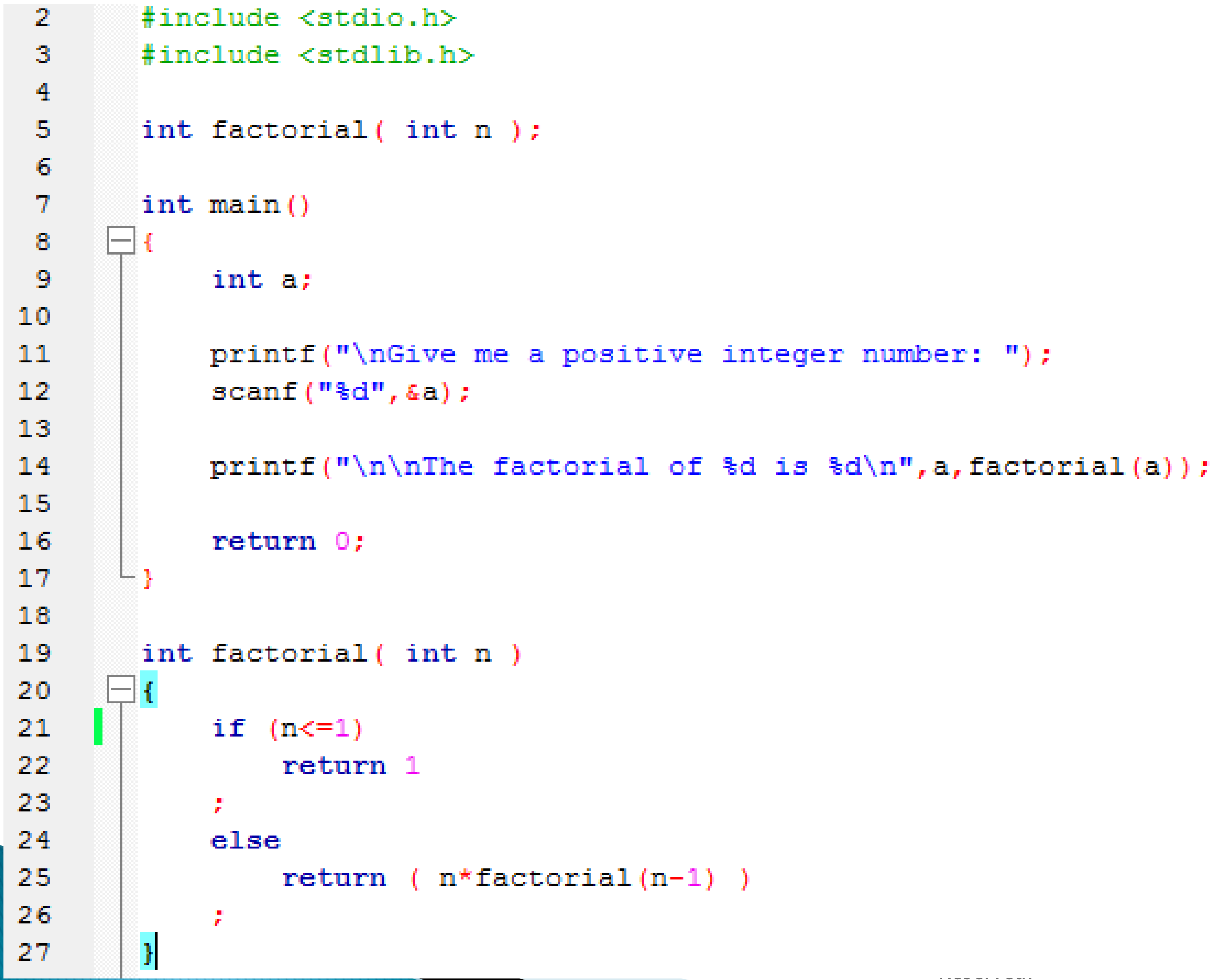
- If `number` is greater than 1, the statement

```
return number * factorial( number - 1 );
```

- expresses the problem as the product of `number` and a recursive call to `factorial` evaluating the factorial of `number - 1`.

- The call `factorial( number - 1 )` is a slightly simpler problem than the original calculation `factorial( number )`.

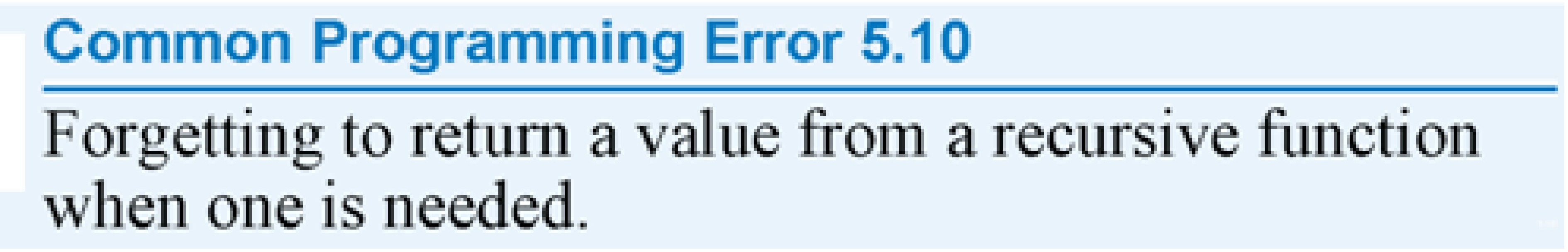
Type this program in Codeblocks



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

int factorial( int n );

int main()
{
    int a;

    printf("\nGive me a positive integer number: ");
    scanf("%d",&a);

    printf("\n\nThe factorial of %d is %d\n",a,factorial(a));

    return 0;
}

int factorial( int n )
{
    if (n<=1)
        return 1
    ;
    else
        return ( n*factorial(n-1) )
    ;
}
```

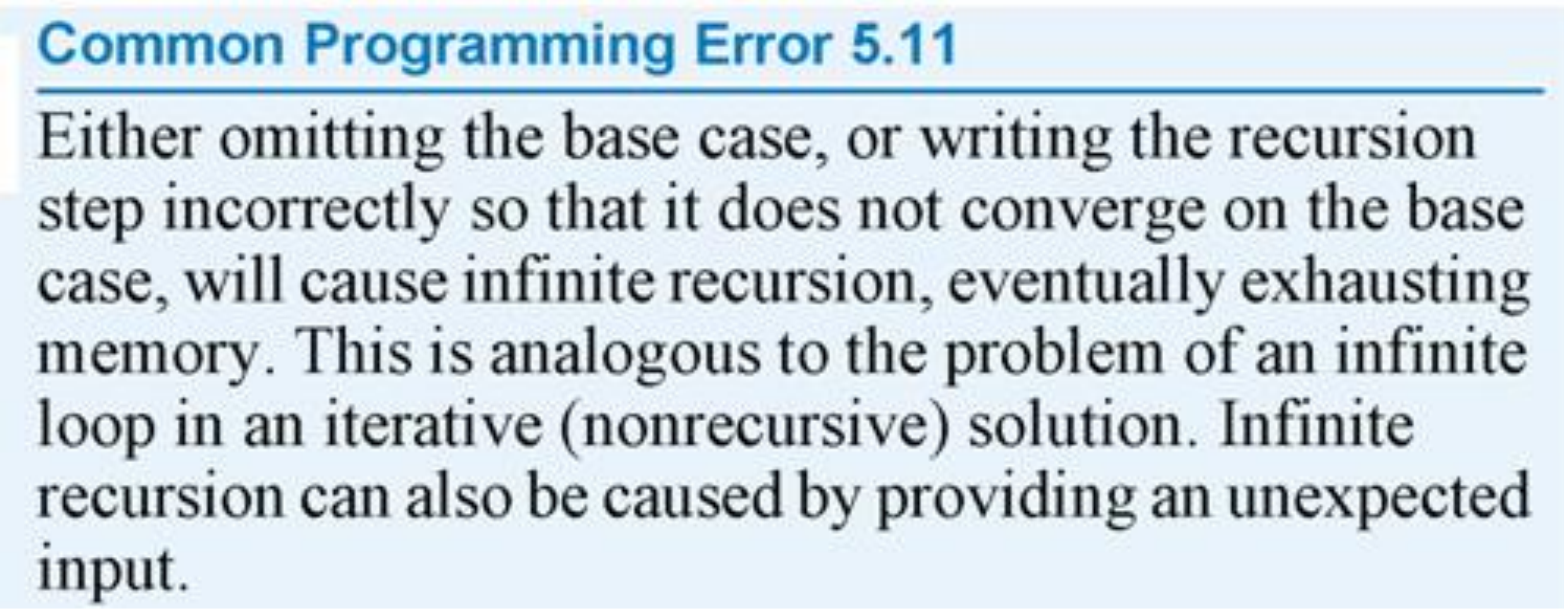
Recursion (Cont.)

- ***Base case***
- The recursive `factorial` function first tests whether a *terminating condition* is true, i.e., whether `number` is less than or equal to 1.



Common Programming Error 5.10

Forgetting to return a value from a recursive function when one is needed.



Common Programming Error 5.11

Either omitting the base case, or writing the recursion step incorrectly so that it does not converge on the base case, will cause infinite recursion, eventually exhausting memory. This is analogous to the problem of an infinite loop in an iterative (nonrecursive) solution. Infinite recursion can also be caused by providing an unexpected input.

Recursion vs. Iteration

- Let’s compare the two approaches and discuss why you might choose one approach over the other in a particular situation.
    - Both iteration and recursion are based on a control structure: Iteration uses a *repetition structure*; recursion uses a *selection structure*.
    - Both iteration and recursion involve repetition: Iteration explicitly uses a *repetition structure*; recursion achieves repetition through *repeated function calls*.

Recursion vs. Iteration (Cont.)

- Iteration and recursion each involve a *termination test*: Iteration terminates when the *loop-continuation condition fails*; recursion when a *base case is recognized*.

- Iteration with counter-controlled repetition and recursion each *gradually approach termination*:

- *Iteration* keeps modifying a counter until the counter assumes a value that makes the *loop-continuation condition fail*;
- *Recursion* keeps producing simpler versions of the original problem until the base case is reached.

Recursion vs. Iteration (Cont.)

- Both iteration and recursion can occur *infinitely*:

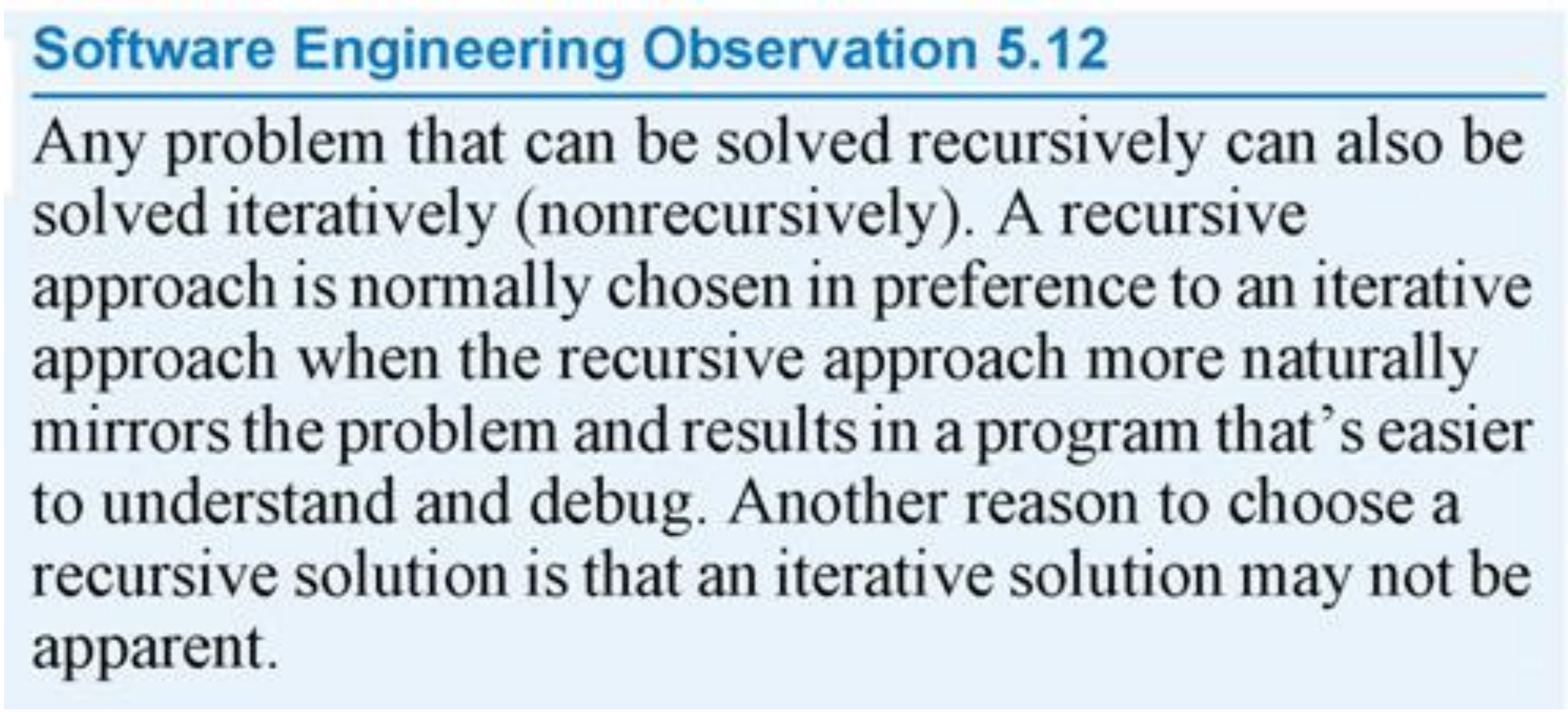
- An *infinite loop* occurs with iteration if the loop-continuation test never becomes false;

- An *infinite recursion* occurs if the recursion step does *not* reduce the problem each time in a manner that converges on the base case, thus producing a stack overflow error.

Recursion negatives

- It *repeatedly* invokes the mechanism, and consequently the *overhead, of function calls*.
- This can be expensive in both processor time and memory space.
- Each recursive call causes *another copy* of the function to be created; this can consume *considerable memory*.

Why choose Recursion?



Software Engineering Observation 5.12

Any problem that can be solved recursively can also be solved iteratively (nonrecursively). A recursive approach is normally chosen in preference to an iterative approach when the recursive approach more naturally mirrors the problem and results in a program that's easier to understand and debug. Another reason to choose a recursive solution is that an iterative solution may not be apparent.

Self Review Exercise – Fibonacci Series

- The Fibonacci series
- 0, 1, 1, 2, 3, 5, 8, 13, 21, …

- It begins with 0 and 1 and has the property that each subsequent Fibonacci number is the sum of the previous two Fibonacci numbers.

- The series above represents the Fibonacci values of the following numbers
- 0, 1, 2, 3, 4, 5, 6, 7, 8, … respectively

Self Review Exercise – Fibonacci Series

- The Fibonacci series may be defined recursively as follows:
- fibonacci(0) = 0
- fibonacci(1) = 1
- fibonacci(n) = fibonacci(n – 1) + fibonacci(n – 2)

- Write a program that calculates the nth Fibonacci number recursively using function fibonacci.

- Notice that Fibonacci numbers tend to become large quickly. use the data type int for the parameter type and the data type unsigned int for the return type in function fibonacci.