5

C Functions

5.12 Storage Classes

- **Automatic storage**
  - **Object created and destroyed within its block**
  - **`auto`: default for local variables**

```
auto double x, y;
```

- **Static storage**
  - **Variables exist for entire program execution**
  - **Default value of zero**
  - **`static`: local variables defined in functions.**
    - **Keep value after function ends**
    - **Only known in their own function**
  - **`extern`: default for global variables and functions**
    - **Known in any function**

5.13 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.)**

5.13 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**

5

Software Engineering Observation 5.11

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, use of global variables should be avoided except in certain situations with unique performance requirements (as discussed in Chapter 14).

6

Common Programming Error 5.14

Accidentally using the same name for an identifier in an inner block as is used for an identifier in an outer block, when in fact you want the identifier in the outer block to be active for the duration of the inner block.

Outline

fig05_12.c

(1 of 4)

```
1  /* Fig. 5.12: fig05_12.c
2     A scoping example */
3  #include <stdio.h>
4
5  void useLocal( void );       /* function prototype */
6  void useStaticLocal( void ); /* function prototype */
7  void useGlobal( void );      /* function prototype */
8
9  int x = 1; /* global variable */
10
11 /* function main begins program execution */
12 int main( void )
13 {
14    int x = 5; /* local variable to main */
15
16    printf("local x in outer scope of main is %d\n", x );
17
18    { /* start new scope */
19       int x = 7; /* local variable to new scope */
20
21       printf( "local x in inner scope of main is %d\n", x );
22    } /* end new scope */
23
```

Global variable with file scope

Variable with block scope

Variable with block scope

Outline

fig05_12.c

(2 of 4)

```
24    printf( "local x in outer scope of main is %d\n", x );
25
26    useLocal();       /* useLocal has automatic local x */
27    useStaticLocal(); /* useStaticLocal has static local x */
28    useGlobal();      /* useGlobal uses global x */
29    useLocal();       /* useLocal reinitializes automatic local x */
30    useStaticLocal(); /* static local x retains its prior value */
31    useGlobal();      /* global x also retains its value */
32
33    printf( "\nlocal x in main is %d\n", x );
34
35    return 0; /* indicates successful termination */
36
37 } /* end main */
38
39 /* useLocal reinitializes local variable x during each call */
40 void useLocal( void )
41 {
42    int x = 25;  /* initialized each time useLocal is called */
43
44    printf( "\nlocal x in useLocal is %d after entering useLocal\n", x );
45    x++;
46    printf( "local x in useLocal is %d before exiting useLocal\n", x );
47 } /* end function useLocal */
48
```

Variable with block scope

```
/* 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 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 */
```

Static variable with block scope

Global variable

Outline

fig05_12.c

(3 of 4)


```
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
```


Outline

fig05_12.c

(4 of 4)

5.14 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**

5.14 Recursion

- **Example: factorials**
  - `5! = 5 * 4 * 3 * 2 * 1`
  - **Notice that**
    - `5! = 5 * 4!`
    - `4! = 4 * 3! ...`
  - **Can compute factorials recursively**
  - **Solve base case (`1! = 0! = 1`) then plug in**
    - `2! = 2 * 1! = 2 * 1 = 2;`
    - `3! = 3 * 2! = 3 * 2 = 6;`

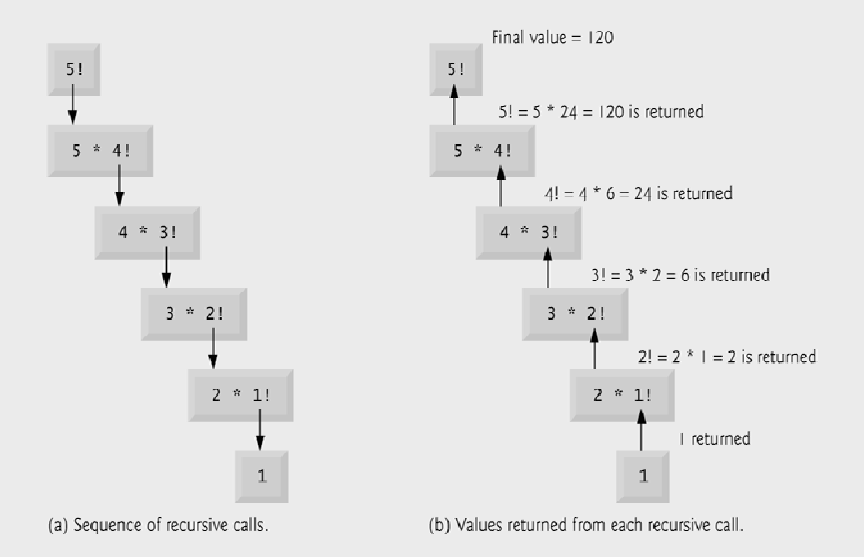

Fig. 5.13 | Recursive evaluation of 5!.

Outline

fig05_14.c

(1 of 2)

```
1  /* Fig. 5.14: fig05_14.c
2     Recursive factorial function */
3  #include <stdio.h>
4
5  long factorial( long number ); /* function prototype */
6
7  /* function main begins program execution */
8  int main( void )
9  {
10    int i; /* counter */
11
12    /* loop 11 times; during each iteration, calculate
13       factorial( i ) and display result */
14    for ( i = 0; i <= 10; i++ ) {
15       printf( "%2d! = %ld\n", i, factorial( i ) );
16    } /* end for */
17
18    return 0; /* indicates successful termination */
19
20 } /* end main */
21
```

Outline

fig05_14.c

(2 of 2)

```
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    else { /* recursive step */
30       return ( number * factorial( number - 1 ) );
31    } /* end else */
32
33 } /* end function factorial */
```

```
 0! = 1
 1! = 1
 2! = 2
 3! = 6
 4! = 24
 5! = 120
 6! = 720
 7! = 5040
 8! = 40320
 9! = 362880
10! = 3628800
```

Common Programming Error 5.16

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.

5.15 Example Using Recursion: Fibonacci Series

- **Fibonacci series: 0, 1, 1, 2, 3, 5, 8...**
  - **Each number is the sum of the previous two**
  - **Can be solved recursively:**
    - `fib( n ) = fib( n - 1 ) + fib( n – 2 )`
  - **Code for the `fibonacci` function**

```
long fibonacci( long n )
{
  if (n == 0 || n == 1)  // base case
     return n;
  else
     return fibonacci( n - 1) +
        fibonacci( n – 2 );
}
```

Outline

fig05_15.c

(1 of 4)

```
1  /* Fig. 5.15: fig05_15.c
2     Recursive fibonacci function */
3  #include <stdio.h>
4
5  long fibonacci( long n ); /* function prototype */
6
7  /* function main begins program execution */
8  int main( void )
9  {
10    long result; /* fibonacci value */
11    long number; /* number input by user */
12
13    /* obtain integer from user */
14    printf( "Enter an integer: " );
15    scanf( "%ld", &number );
16
17    /* calculate fibonacci value for number input by user */
18    result = fibonacci( number );
19
20    /* display result */
21    printf( "Fibonacci( %ld ) = %ld\n", number, result );
22
23    return 0; /* indicates successful termination */
24
25 } /* end main */
26
```

(continued from previous slide…)

```
Enter an integer: 3
Fibonacci( 3 ) = 2

Enter an integer: 4
Fibonacci( 4 ) = 3

Enter an integer: 5
Fibonacci( 5 ) = 5
```

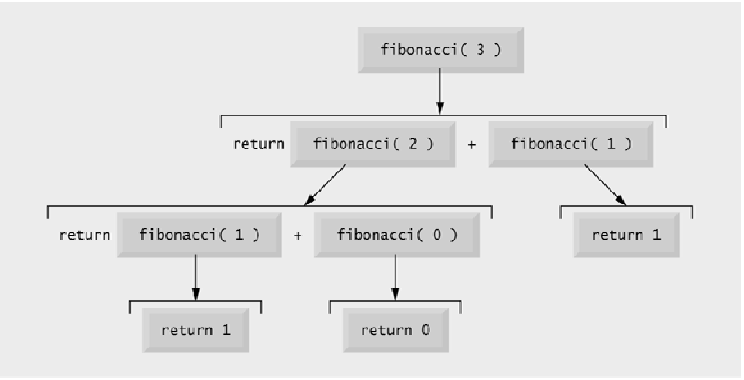

Fig. 5.16 | Set of recursive calls for fibonacci(3).

Outline

fig05_15.c

(3 of 4)

5.16 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)**
  - **Avoid using recursion in performance situations**
    - **Recursive calls take time and consume additional memory**

18

C++ as a Better C; Introducing Object Technology

18.6 Inline Functions

- **Inline functions**
  - **Reduce function call overhead—especially for small functions**
  - **Qualifier `inline` before a function's return type in the function definition**
    - **"Advises" the compiler to generate a copy of the function's code in place (when appropriate) to avoid a function call**
  - **Trade-off of inline functions**
    - **Multiple copies of the function code are inserted in the program (often making the program larger)**
  - **The compiler can ignore the `inline` qualifier and typically does so for all but the smallest functions**

Performance Tip 18.2

Using `inline` functions can reduce execution time but may increase program size.

→ The `inline` qualifier should be used only with small, frequently used functions.

Outline

fig18_03.cpp

(1 of 2)

```
1  // Fig. 18.3: fig18_03.cpp
2  // Using an inline function to calculate the volume of a cube.
3  #include <iostream>
4  using std::cout;
5  using std::cin;
6  using std::endl;
7
8  // Definition of inline function cube. Definition of function appears
9  // before function is called, so a function prototype is not required.
10 // First line of function definition acts as the prototype.
11 inline double cube( const double side )
12 {
13    return side * side * side; // calculate the cube of side
14 } // end function cube
15
16 int main()
17 {
18    double sideValue; // stores value entered by user
19
```

`inline` qualifier

Complete function definition so the compiler knows how to expand a **`cube`** function call into its inlined code.

Outline

fig18_03.cpp

(2 of 2)

```
20    for ( int i = 1; i <= 3; i++ )
21    {
22       cout << "\nEnter the side length of your cube: ";
23       cin >> sideValue; // read value from user
24
25       // calculate cube of sideValue and display result
26       cout << "Volume of cube with side "
27          << sideValue << " is " << cube( sideValue ) << endl;
28    }
29
30    return 0; // indicates successful termination
31 } // end main
```

cube function call that could be inlined

```
Enter the side length of your cube: 1.0
Volume of cube with side 1 is 1

Enter the side length of your cube: 2.3
Volume of cube with side 2.3 is 12.167

Enter the side length of your cube: 5.4
Volume of cube with side 5.4 is 157.464
```

18.7 References and Reference Parameters

- **Two ways to pass arguments to functions**
  - **Pass-by-value**
    - **A *copy* of the argument's value is passed to the called function**
    - **Changes to the copy do not affect the original variable's value in the caller**
      - **Prevents accidental side effects of functions**
  - **Pass-by-reference**
    - **Gives called function the ability to access and modify the caller's argument data directly**

Performance Tip 18.3

One disadvantage of pass-by-value is that, if a large data item is being passed, copying that data can take a considerable amount of execution time and memory space.

18.7 References and Reference Parameters (Cont.)

- **Reference Parameter**
  - **An alias for its corresponding argument in a function call**
  - **& placed after the parameter type in the function prototype and function header**
  - **Example**
    - `int &count` **in a function header**
      - **Pronounced as "`count` is a reference to an `int`"**
  - **Parameter name in the body of the called function actually refers to the original variable in the calling function**

Outline

```
1  // Fig. 18.5: fig18_05.cpp
2  // Comparing pass-by-value and pass-by-reference with references.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int squareByValue( int ); // function prototype (value pass)
8  void squareByReference( int & ); // function prototype (reference pass)
9
10 int main()
11 {
12    int x = 2; // value to square using squareByValue
13    int z = 4; // value to square using squareByReference
14
15    // demonstrate squareByValue
16    cout << "x = " << x << " before squareByValue\n";
17    cout << "Value returned by squareByValue: "
18       << squareByValue( x ) << endl;
19    cout << "x = " << x << " after squareByValue\n" <<
20
21    // demonstrate squareByReference
22    cout << "z = " << z << " before squareByReference" << endl;
23    squareByReference( z );
24    cout << "z = " << z << " after squareByReference" << endl;
25    return 0; // indicates successful termination
26 } // end main
```

Function illustrating pass-by-value

Function illustrating pass-by-reference

Variable is simply mentioned by name in both function calls

fig18_05.cpp

(1 of 2)

Outline

```
27
28 // squareByValue multiplies number by itself, stores the
29 // result in number and returns the new value of number
30 int squareByValue( int number )
31 {
32    return number *= number; // caller's argument not modified
33 } // end function squareByValue
34
35 // squareByReference multiplies numberRef by itself and stores the result
36 // in the variable to which numberRef refers in the caller
37 void squareByReference( int &numberRef )
38 {
39    numberRef *= numberRef; // caller's argument modified
40 } // end function squareByReference

x = 2 before squareByValue
Value returned by squareByValue: 4
x = 2 after squareByValue

z = 4 before squareByReference
z = 16 after squareByReference
```

Receives copy of argument in `main`

Receives reference to argument in `main`

Modifies variable in `main`

fig18_05.cpp

(2 of 2)

Performance Tip 18.4

Pass-by-reference is good for performance reasons, because it can eliminate the pass-by-value overhead of copying large amounts of data.

Software Engineering Observation 18.6

Pass-by-reference can weaken security, because the called function can corrupt the caller's data.

Use a constant reference parameter to simulate the appearance and security of pass-by-value and avoid the overhead of passing a copy of the large object. The called function will not be able to modify the object in the caller.

Software Engineering Observation 18.8

For the combined reasons of clarity and performance, many C++ programmers prefer that modifiable arguments be passed to functions by using pointers, small nonmodifiable arguments be passed by value and large nonmodifiable arguments be passed by using references to constants.

Common Programming Error 18.4

Attempting to reassign a previously declared reference to be an alias to another variable is a logic error. The value of the other variable is simply assigned to the variable for which the reference is already an alias.

18.8 Empty Parameter Lists

- **Empty parameter list**
  - **Specified by writing either `void` or nothing at all in parentheses**
  - **For example:**

```
void print();
void print(void);
```

Specifies that function `print` does not take arguments and does not return a value

Portability Tip 18.2

The meaning of an empty function parameter list in C++ is dramatically different than in C. In C, it means all argument checking is disabled (i.e., the function call can pass any arguments it wants). In C++, it means that the function explicitly takes no arguments. Thus, C programs using this feature might cause compilation errors when compiled in C++.

18.9 Default Arguments

- **Default argument**
  - **A default value to be passed to a parameter**
    - **Used when the function call does not specify an argument for that parameter**
  - **Must be the rightmost argument(s) in a function's parameter list**
  - **Should be specified with the first occurrence of the function name**
    - **Typically the function prototype**

Outline

fig18_08.cpp

(1 of 2)

```
1  // Fig. 18.8: fig18_08.cpp
2  // Using default arguments.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  // function prototype that specifies default arguments
8  int boxVolume( int length = 1, int width = 1, int height = 1 );
9
10 int main()
11 {
12    // no arguments--use default values for all dimensions
13    cout << "The default box volume is: " << boxVolume();
14
15    // specify length; default width and height
16    cout << "\n\nThe volume of a box with length 10,\n"
17       << "width 1 and height 1 is: " << boxVolume( 10 );
18
19    // specify length and width; default height
20    cout << "\n\nThe volume of a box with length 10,\n"
21       << "width 5 and height 1 is: " << boxVolume( 10, 5 );
22
23    // specify all arguments
24    cout << "\n\nThe volume of a box with length 10,\n"
25       << "width 5 and height 2 is: " << boxVolume( 10, 5, 2 )
26       << endl;
27    return 0; // indicates successful termination
28 } // end main
```

Default arguments

Calling function with no arguments

Calling function with one argument

Calling function with two arguments

Calling function with three arguments

```
29
30 // function boxVolume calculates the volume of a box
31 int boxVolume( int length, int width, int height )
32 {
33    return length * width * height;
34 } // end function boxVolume

The default box volume is: 1

The volume of a box with length 10,
width 1 and height 1 is: 10

The volume of a box with length 10,
width 5 and height 1 is: 50

The volume of a box with length 10,
width 5 and height 2 is: 100
```

Outline

fig18_08.cpp

Note that default arguments were specified in the function prototype, so they are not specified in the function header

Common Programming Error 18.6

It is a compilation error to specify default arguments in both a function's prototype and header.