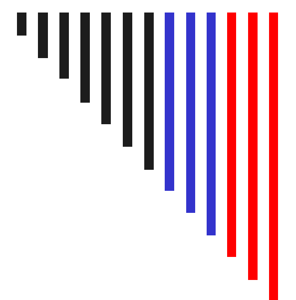


Chapter 9
Pointers

Objectives

- ❑ To understand the concept and use of pointers
- ❑ To be able to declare, define, and initialize pointers
- ❑ To write programs that access data through pointers
- ❑ To use pointers as parameters and return types
- ❑ To understand pointer compatibility, especially regarding pointers to pointers
- ❑ To understand the role of quality in software engineering

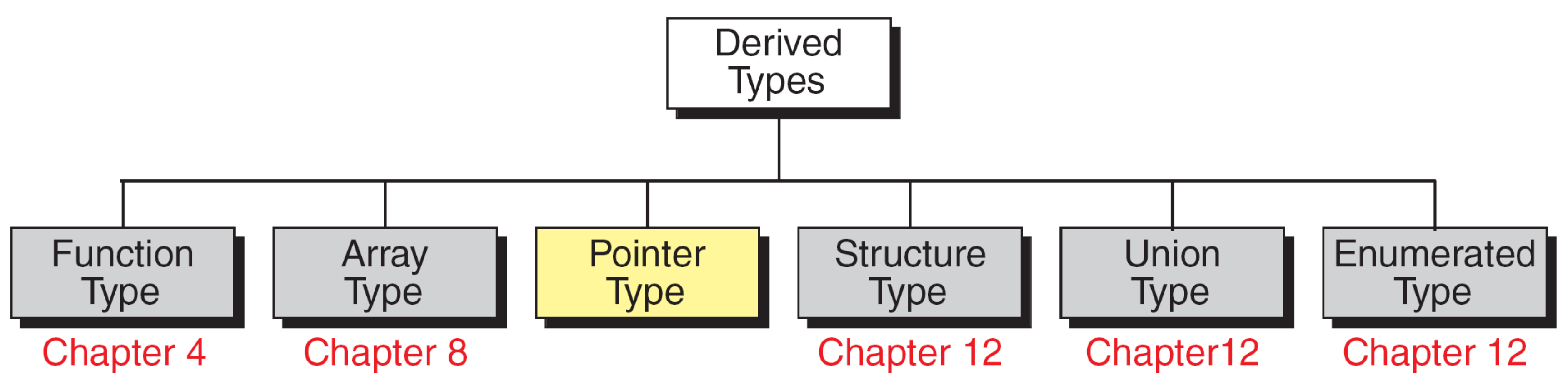


FIGURE 9-1 **Derived Types**

9-1 Introduction

A pointer is a constant or variable that contains an address that can be used to access data. Pointers are built on the basic concept of pointer constants.

<u>Topics discussed in this section:</u>

Pointer Constants
Pointer Values
Pointer Variables
Accessing Variables Through Pointers
Pointer Declaration and Definition
Declaration versus Redirection
Initialization of Pointer Variables

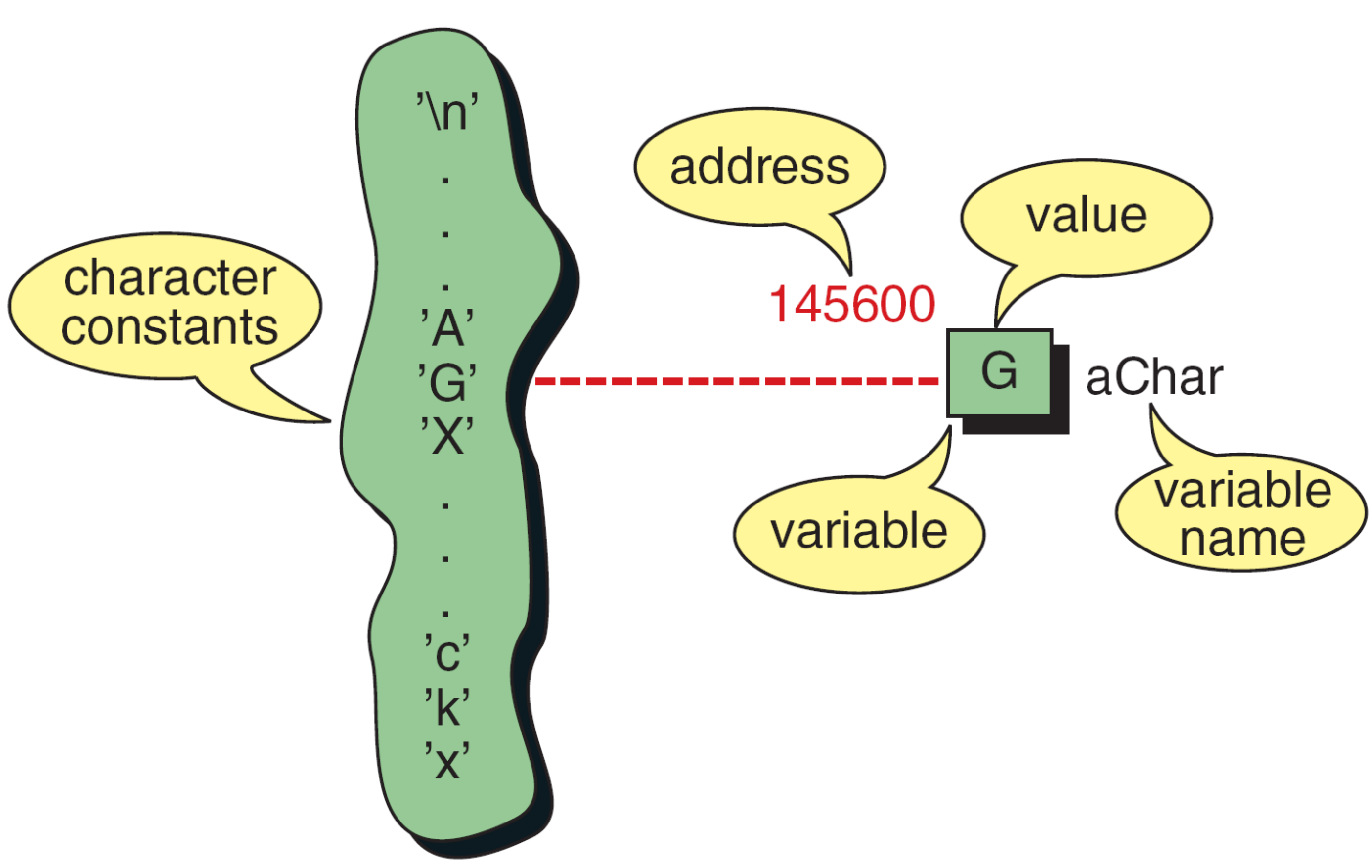


FIGURE 9-2 Character Constants and Variables

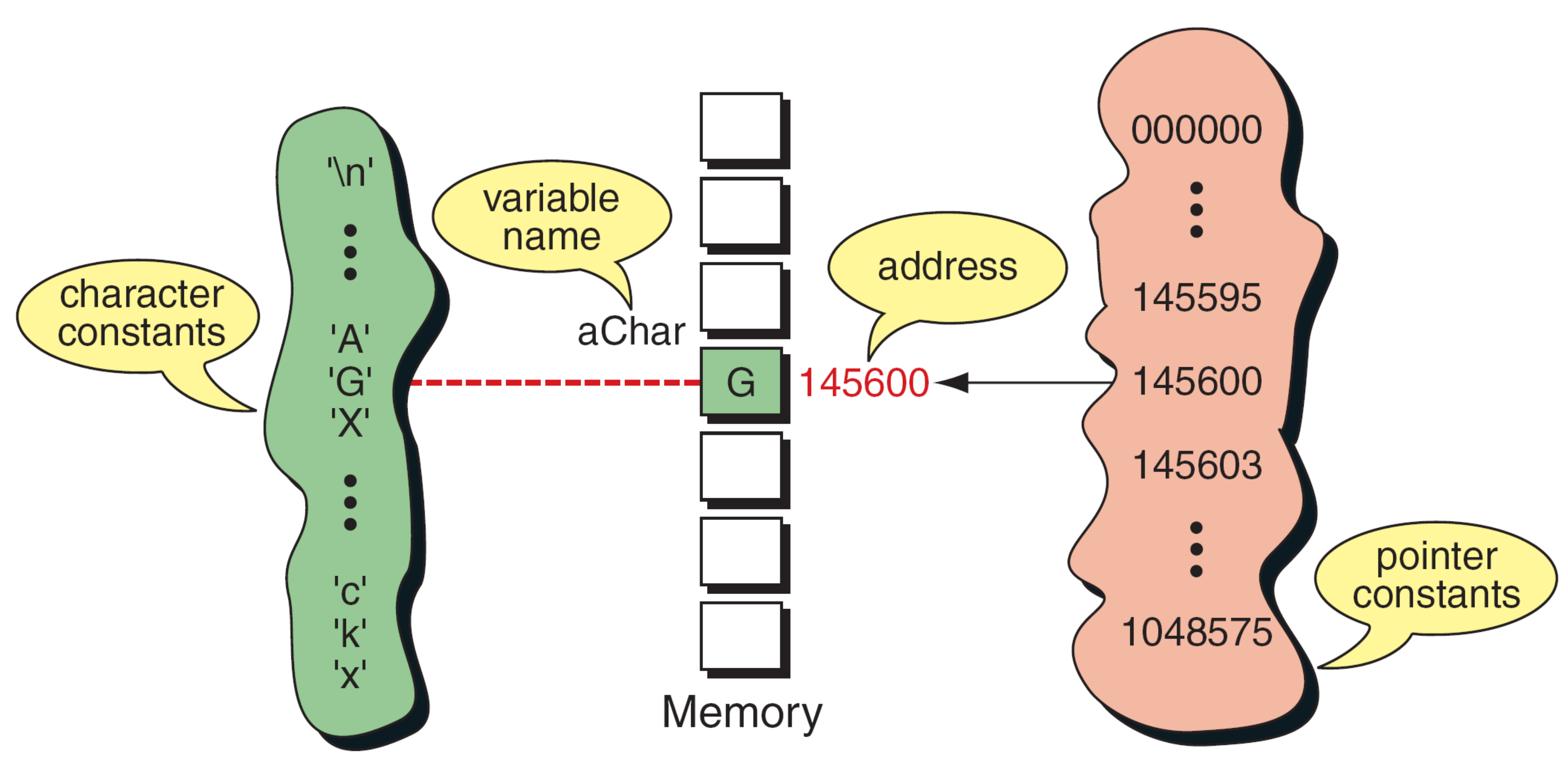


FIGURE 9-3 Pointer Constants

Note

Pointer constants, drawn from the set of addresses for a computer, exist by themselves. We cannot change them; we can only use them.

Note

An address expression, one of the expression types in the unary expression category, consists of an ampersand (&) and a variable name.

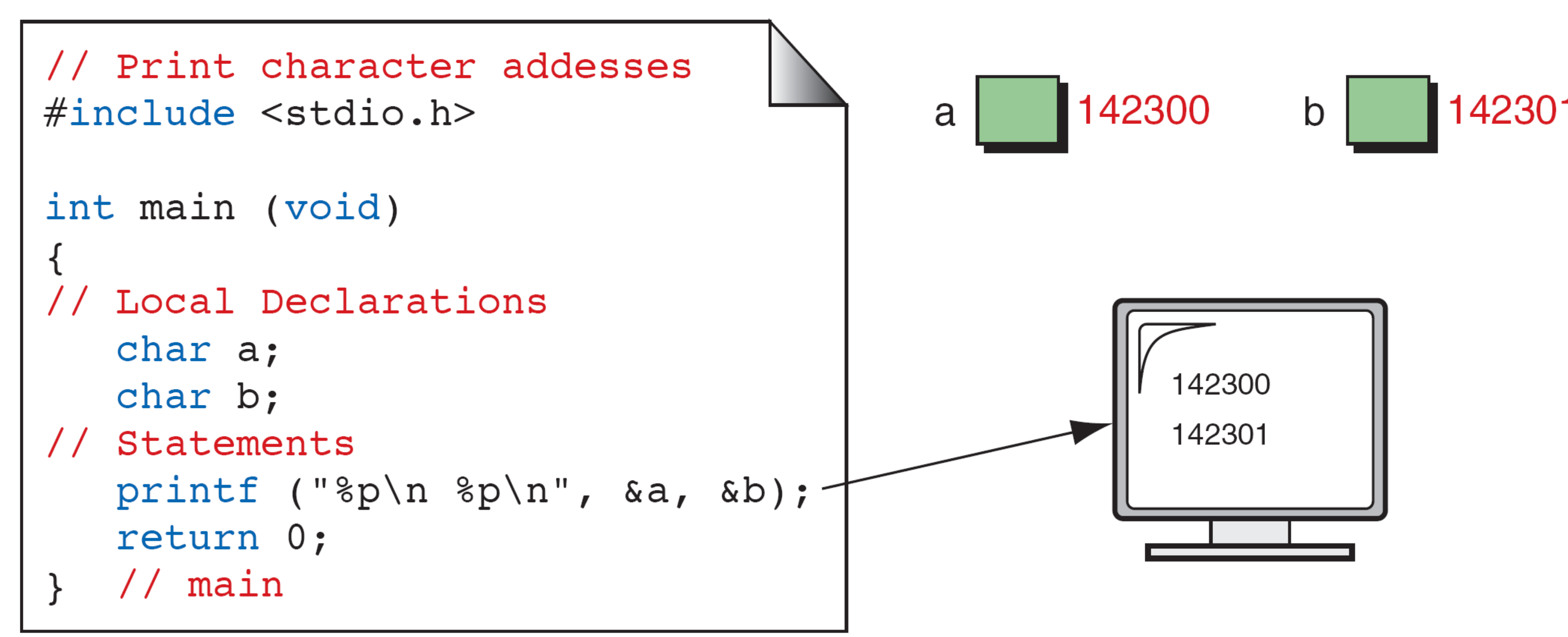


FIGURE 9-4 **Print Character Addresses**

Note

A variable’s address is the first byte occupied by the variable.

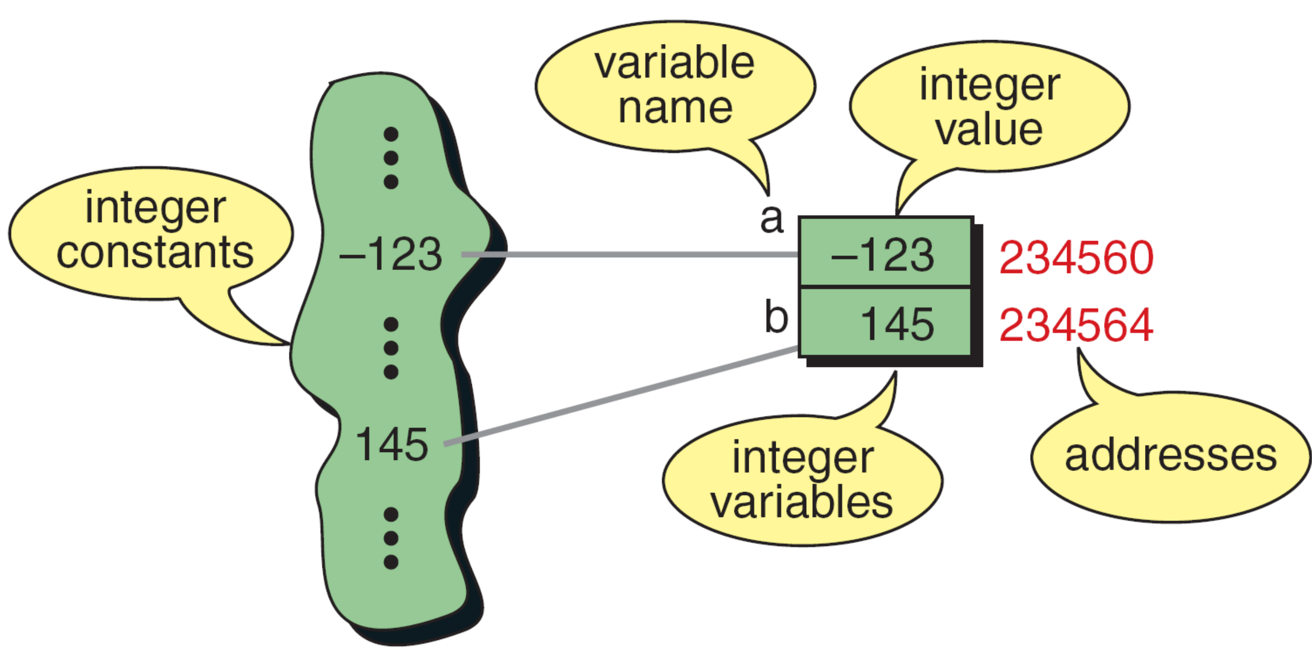
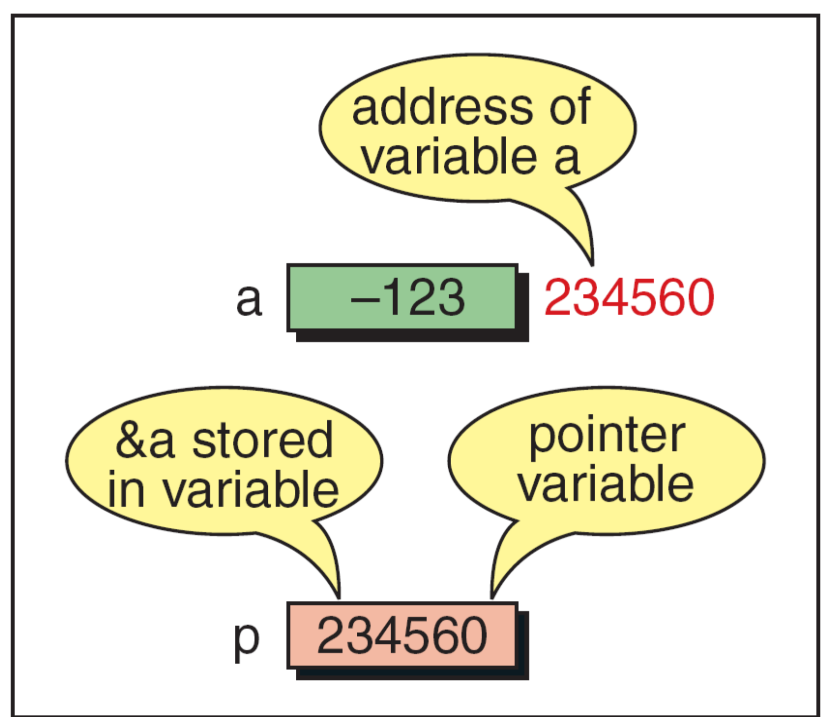
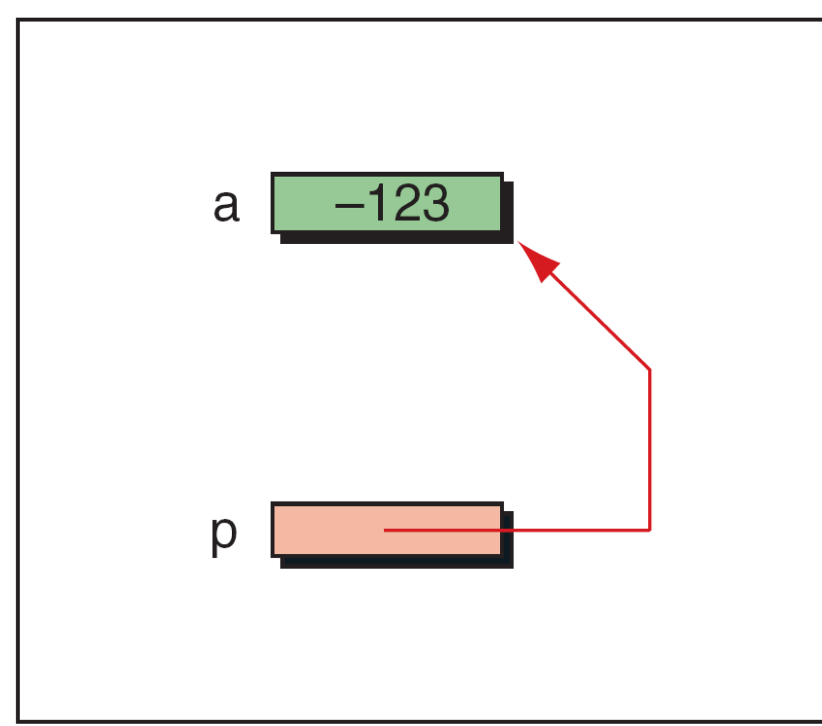


FIGURE 9-5 Integer Constants and Variables

```
int a = -123;
int* p;
p = &a;
```



Physical representation



Logical representation

FIGURE 9-6 Pointer Variable

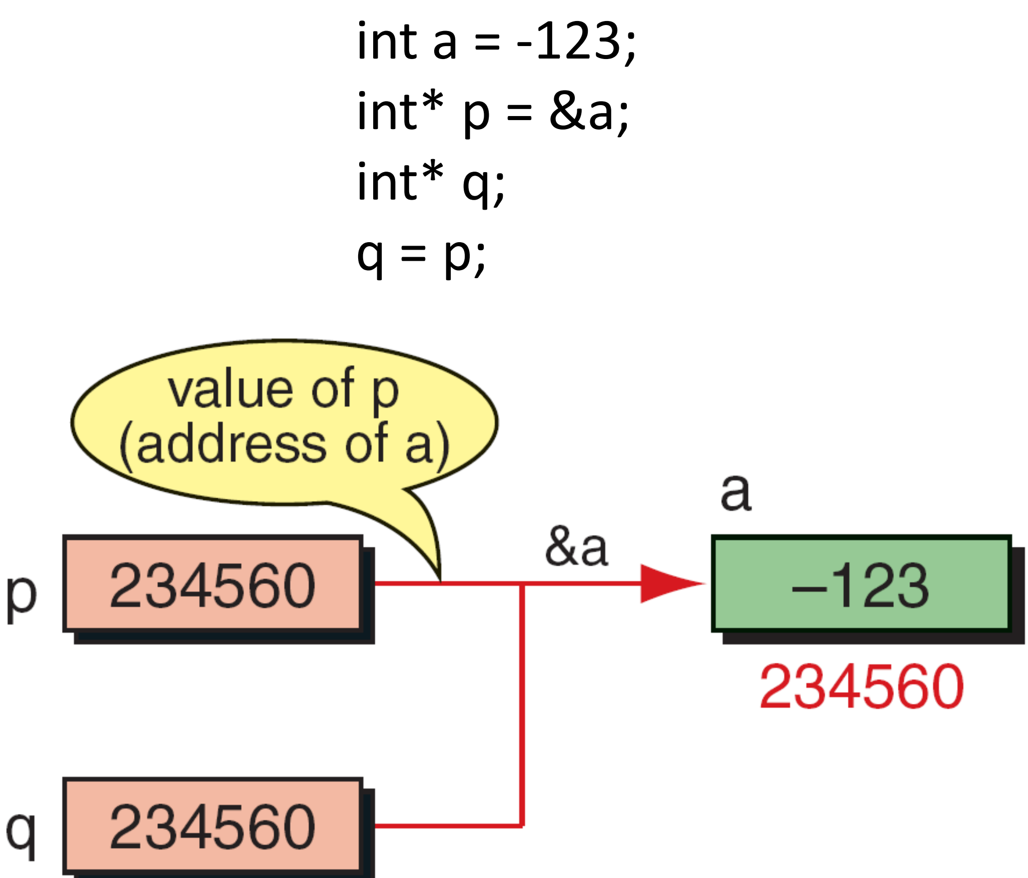


FIGURE 9-7 Multiple Pointers to a Variable

Note

A pointer that points to no variable contains the special null-pointer constant, NULL.

```
int* p = NULL;
```

Note

An indirect expression, one of the expression types in the unary expression category, is coded with an asterisk (*) and an identifier.

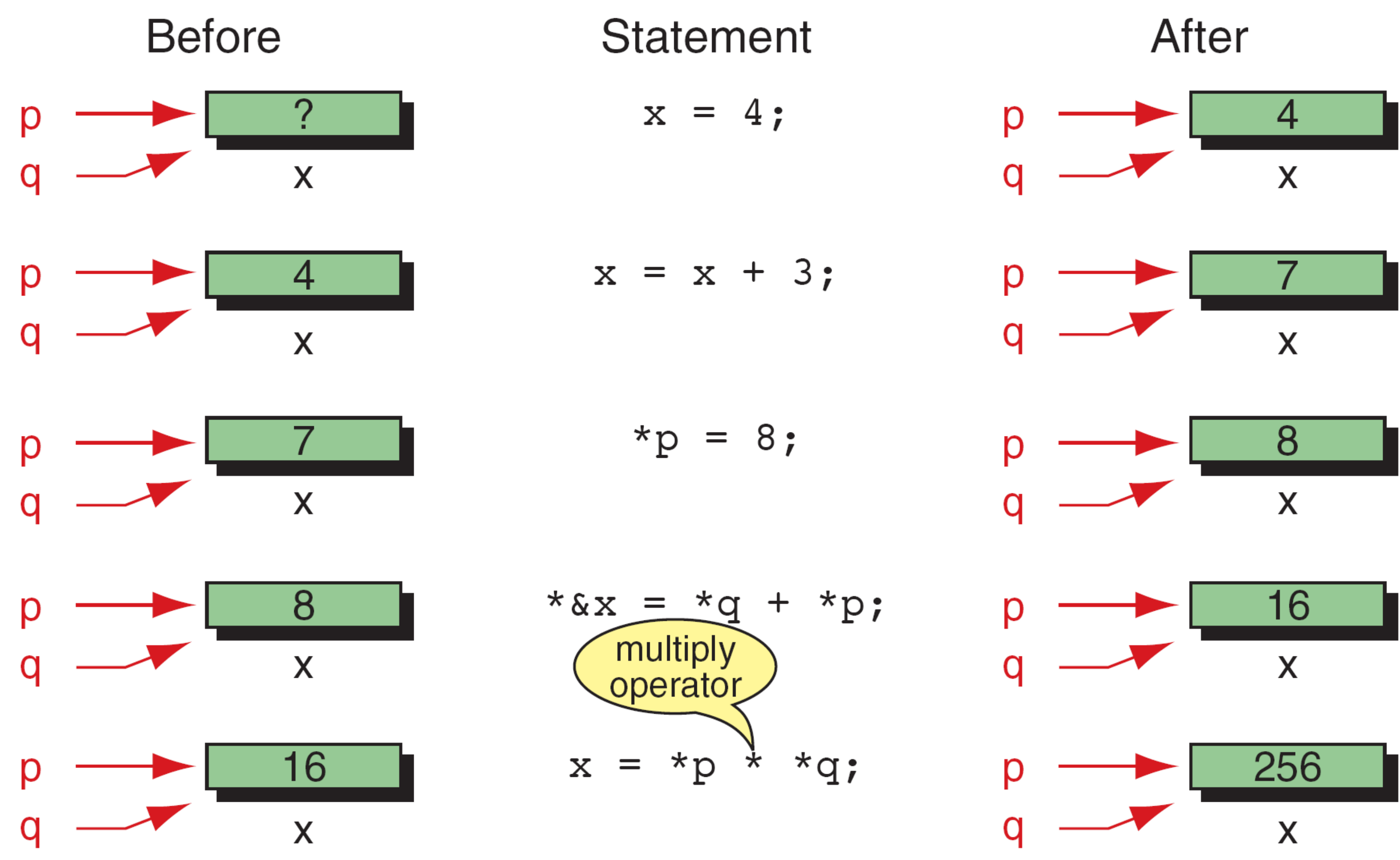


FIGURE 9-8 **Accessing Variables Through Pointers**

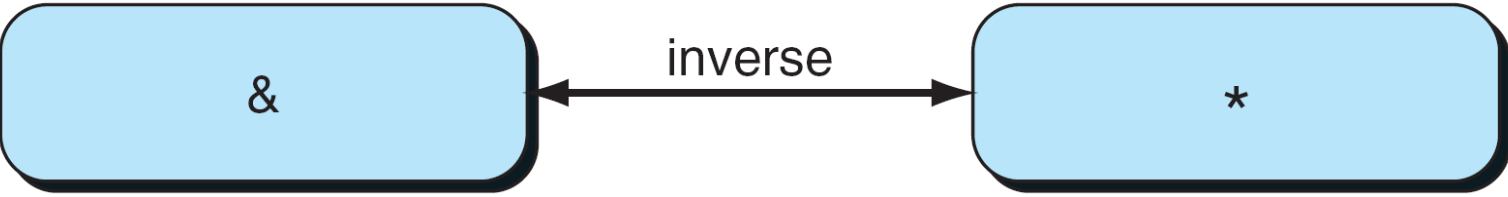


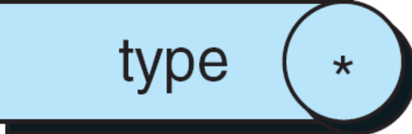
FIGURE 9-9 Address and Indirection Operators

data declaration

type

identifier

pointer declaration



identifier

FIGURE 9-10 Pointer Variable Declaration

```
char a;     Z          char* p;
int n;      15         int* q;
float x;    3.3        float* r;
```

FIGURE 9-11 Declaring Pointer Variables

PROGRAM 9-1 Demonstrate Use of Pointers

```
1   /* Demonstrate pointer use
2         Written by:
3         Date:
4   */
5   #include <stdio.h>
6
7   int main (void)
8   {
9   // Local Declarations
10     int  a;
11     int* p;
12
13  // Statements
14     a = 14;
15     p = &a;
16
17     printf("%d %p\n", a, &a);
```

a 14
135760
p 135760

PROGRAM 9-1 Demonstrate Use of Pointers

```
18     printf("%p %d %d\n",
19                 p, *p, a);
20
21     return 0;
22  }  // main
```

```
Results:
14 00135760
00135760 14 14
```

Declaration versus Redirection

- int* pa;
- int* pb;

- sum = *pa + *pb;

- Address operator &

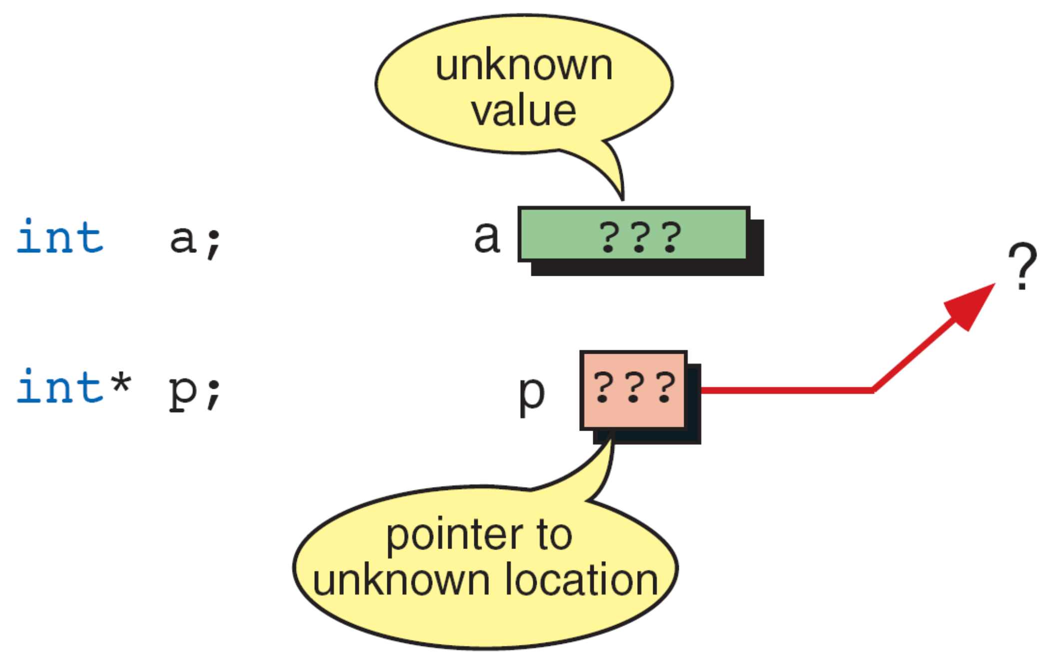


FIGURE 9-12 Uninitialized Pointers

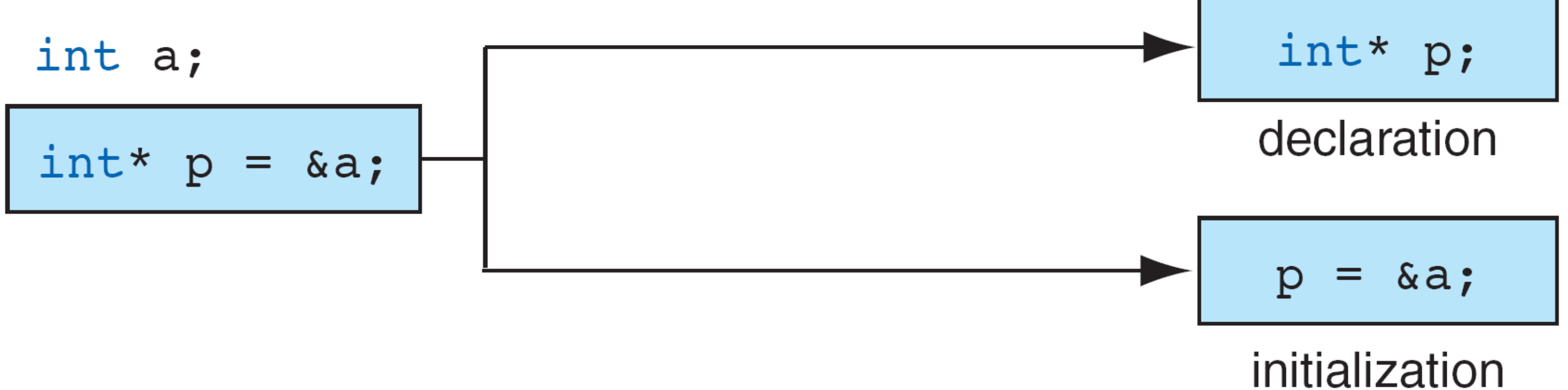


FIGURE 9-13 **Initializing Pointer Variables**

PROGRAM 9-2 Fun with Pointers

```
 1  /* Fun with pointers
 2        Written by:
 3        Date:
 4  */
 5  #include <stdio.h>
 6
 7  int main (void)
 8  {
 9  // Local Declarations
10     int  a;
11     int  b;
12     int  c;
13     int* p;
14     int* q;
15     int* r;
16
17  // Statements
18     a = 6;
19     b = 2;
20     p = &b;
21
```

PROGRAM 9-2 Fun with Pointers

```
22      q = p;
23      r = &c;
24
25      p  = &a;
26      *q = 8;
27
28      *r = *p;
29
30      *r = a + *q + *&c;
31
32      printf("%d %d %d \n",
33               a, b, c);
```

PROGRAM 9-2 Fun with Pointers

```
34        printf("%d %d %d",
35                *p, *q, *r);
36        return 0;
37    }   // main
```

```
Results:
6 8 20
6 8 20
```

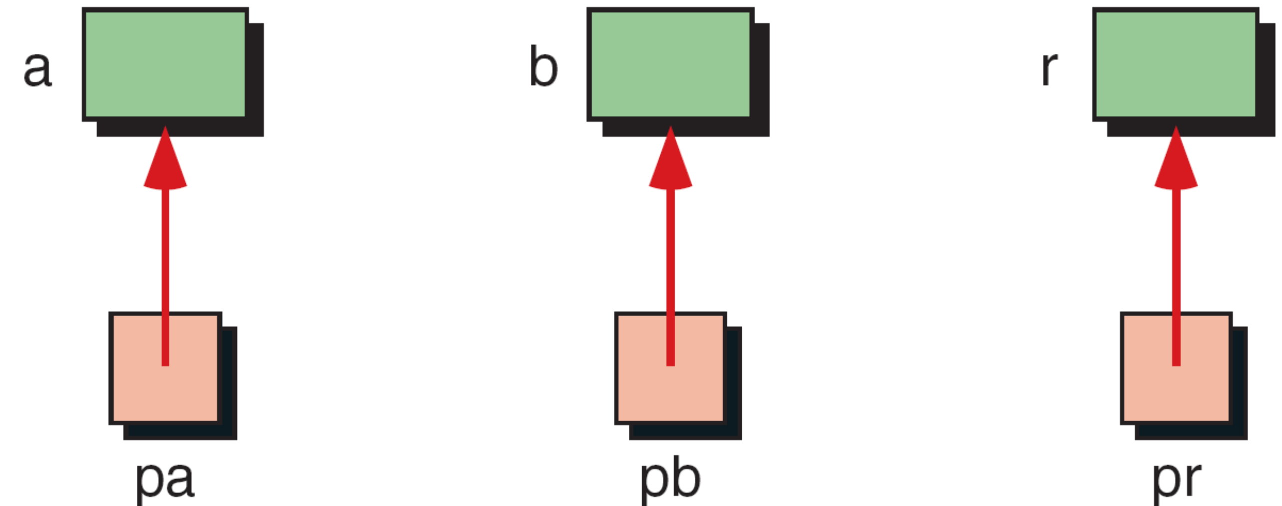


FIGURE 9-14 Add Two Numbers Using Pointers

PROGRAM 9-3 Add Two Numbers Using Pointers

```
 1  /* This program adds two numbers using pointers to
 2     demonstrate the concept of pointers.
 3        Written by:
 4        Date:
 5  */
 6  #include <stdio.h>
 7
 8  int main (void)
 9  {
10  // Local Declarations
11     int  a;
12     int  b;
13     int  r;
14     int* pa = &a;
15     int* pb = &b;
16     int* pr = &r;
17
```

PROGRAM 9-3 Add Two Numbers Using Pointers

```
18 // Statements
19    printf("Enter the first number : ");
20    scanf ("%d", pa);
21    printf("Enter the second number: ");
22    scanf ("%d", pb);
23    *pr = *pa + *pb;
24    printf("\n%d + %d is %d", *pa, *pb, *pr);
25    return 0;
26 } // main
```

```
Results:
   Enter the first number : 15
   Enter the second number: 51

   15 + 51 is 66
```

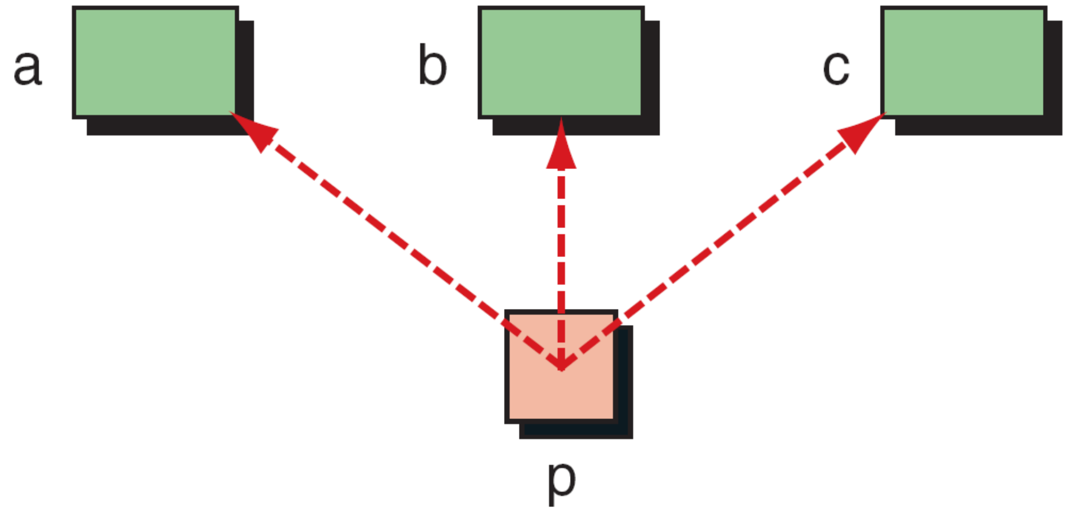


FIGURE 9-15 **Demonstrate Pointer Flexibility**

PROGRAM 9-4 Using One Pointer for Many Variables

```
1   /* This program shows how the same pointer can point to
2      different data variables in different statements.
3         Written by:
4         Date:
5   */
6   #include <stdio.h>
7
8   int main (void)
9   {
10  // Local Declarations
11     int  a;
12     int  b;
13     int  c;
14     int* p;
15
16  // Statements
17     printf("Enter three numbers and key return: ");
18     scanf ("%d %d %d", &a, &b, &c);
```

PROGRAM 9-4 Using One Pointer for Many Variables

```
19     p = &a;
20     printf("%3d\n", *p);
21     p = &b;
22     printf("%3d\n", *p);
23     p = &c;
24     printf("%3d\n", *p);
25     return 0;
26  }  // main
```

```
Results:
Enter three numbers and key return: 10 20 30
 10
 20
 30
```

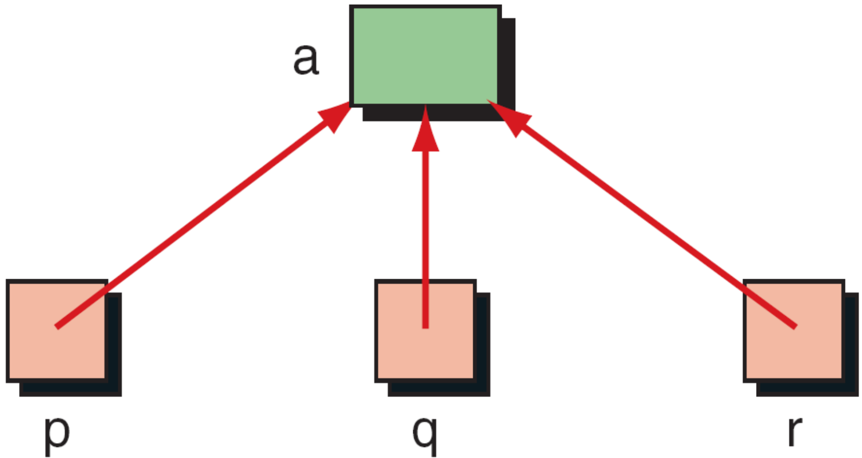


FIGURE 9-16 **One Variable with Many Pointers**

PROGRAM 9-5 Using A Variable with Many Pointers

```
1  /* This program shows how we can use different pointers
2     to point to the same data variable.
3        Written by:
4        Date:
5  */
6  #include <stdio.h>
7
8  int main (void)
9  {
10 // Local Declarations
11    int  a;
12    int* p = &a;
13    int* q = &a;
14    int* r = &a;
15
16 // Statements
17    printf("Enter a number: ");
18    scanf ("%d", &a);
```

PROGRAM 9-5 Using A Variable with Many Pointers

```
19        printf("%d\n", *p);
20        printf("%d\n", *q);
21        printf("%d\n", *r);
22
23        return 0;
24    }   // main
```

```
Results:
Enter a number: 15
15
15
15
```

9-2 Pointers for Inter-function Communication

One of the most useful applications of pointers is in functions. When we discussed functions in Chapter 4, we saw that C uses the pass-by-value for downward communication. For upward communication, we normally pass an address. In this section, we fully develop the bi-directional communication.

Topics discussed in this section:

Passing Addresses
Functions Returning Pointers

```c
// Function Declarations
void exchange (int x, int y);

int main (void)
{
  int a = 5;
  int b = 7;
  exchange (a, b);
  printf("%d %d\n", a, b);
  return 0;
} // main
```

```c
void exchange (int x, int y)
{
  int temp;

  temp = x;
  x    = y;
  y    = temp;
  return;
} // exchange
```

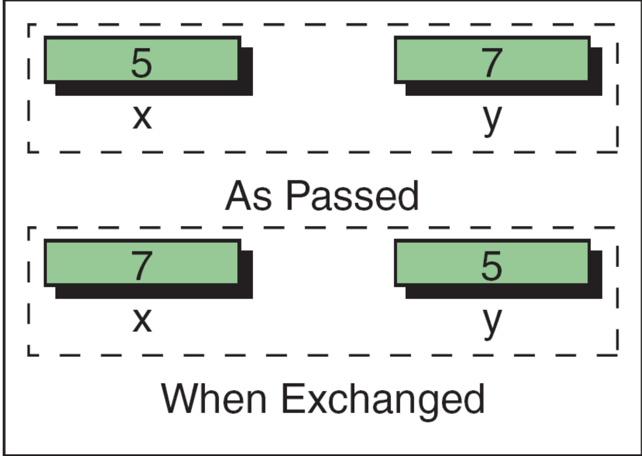


FIGURE 9-17 An Unworkable Exchange

```
// Function Declaration
void exchange (int*, int*);

int main (void)
{
  int a = 5;
  int b = 7;

  exchange (&a, &b);
  printf("%d %d\n", a, b);
  return 0;
} // main
```

```
void exchange (int* px, int* py)
{
  int temp;

  temp = *px;
  *px  = *py;
  *py  = temp;
  return;
} // exchange
```


FIGURE 9-18 Exchange Using Pointers

Note

Every time we want a called function to have access to a variable in the calling function, we pass the address of that variable to the called function and use the indirection operator to access it.

```c
// Prototype Declarations
int* smaller (int* p1, int* p2);

int main (void)
  …
  int  a;
  int  b;
  int* p;
  …
  scanf ( "%d %d", &a, &b );
  p = smaller (&a, &b);
  …
```

```c
int* smaller (int* px, int* py)
{
  return (*px < *py ? px : py);
} // smaller
```

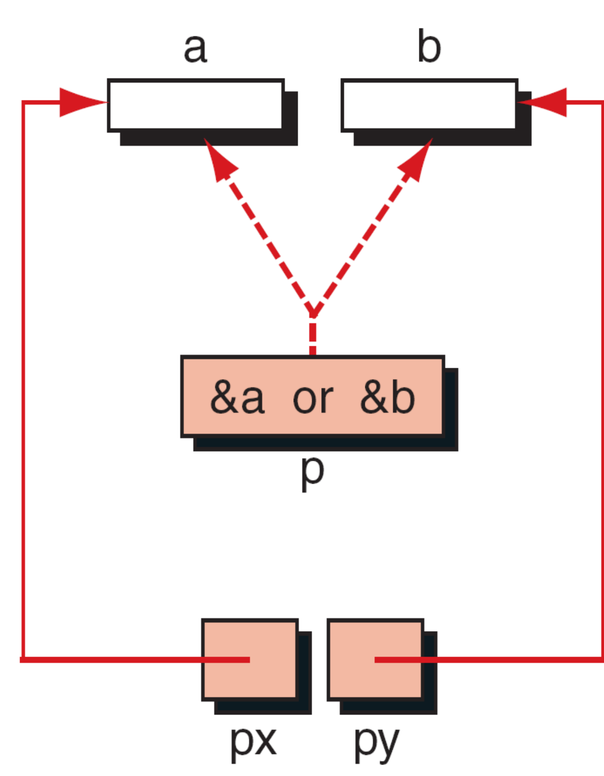


FIGURE 9-19 **Functions Returning Pointers**

Note

It is a serious error to return a pointer to a local variable.

9-3 Pointers to Pointers

So far, all our pointers have been pointing directly to data. It is possible—and with advanced data structures often necessary—to use pointers that point to other pointers. For example, we can have a pointer pointing to a pointer to an integer.

```c
// Local Declarations
int   a;
int*  p;
int** q;
```



```c
// Statements
a = 58;
p = &a;
q = &p;
printf(" %3d",   a);
printf(" %3d",  *p);
printf(" %3d", **q);
```

FIGURE 9-20 Pointers to Pointers

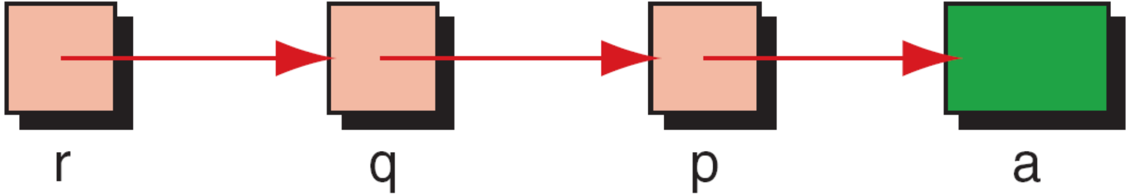


FIGURE 9-21 Using Pointers to Pointers

PROGRAM 9-6 Using pointers to pointers

```
1   /* Show how pointers to pointers can be used by different
2      scanf functions to read data to the same variable.
3         Written by:
4         Date:
5   */
6   #include <stdio.h>
7
8   int main (void)
9   {
10  // Local Declarations
11     int    a;
12     int*   p;
13     int**  q;
14     int*** r;
15
16  // Statements
17     p = &a;
```

PROGRAM 9-6 Using pointers to pointers

```
18        q = &p;
19        r = &q;
20     }
21        printf("Enter a number: ");
22        scanf ("%d", &a);                       // Using a
23        printf("The number is : %d\n", a);
24
25        printf("\nEnter a number: ");
26        scanf ("%d", p);                        // Using p
27        printf("The number is : %d\n", a);
28
29        printf("\nEnter a number: ");
30        scanf ("%d", *q);                       // Using q
31        printf("The number is : %d\n", a);
32
33        printf("\nEnter a number: ");
34        scanf ("%d", **r);                      // Using r
35        printf("The number is : %d\n", a);
36
37        return 0;
38     }  // main
```

PROGRAM 9-6 Using pointers to pointers

```
Results:
Enter a number: 1
The number is : 1

Enter a number: 2
The number is : 2

Enter a number: 3
The number is : 3

Enter a number: 4
The number is : 4
```

9-4 Compatibility

It is important to recognize that pointers have a type associated with them. They are not just pointer types, but rather are pointers to a specific type, such as character. Each pointer therefore takes on the attributes of the type to which it refers in addition to its own attributes.

Topics discussed in this section:

Pointer Size Compatibility
Dereference Type Compatibility
Dereference Level Compatibility

PROGRAM 9-7 Demonstrate Size of Pointers

```
1  /* Demonstrate size of pointers.
2        Written by:
3        Date:
4  */
5  #include <stdio.h>
6
7  int main (void)
8  {
9  // Local Declarations
10    char  c;
11    char* pc;
12    int   sizeofc      = sizeof(c);
13    int   sizeofpc     = sizeof(pc);
14    int   sizeofStarpc = sizeof(*pc);
15
16    int  a;
17    int* pa;
18    int  sizeofa      = sizeof(a);
```

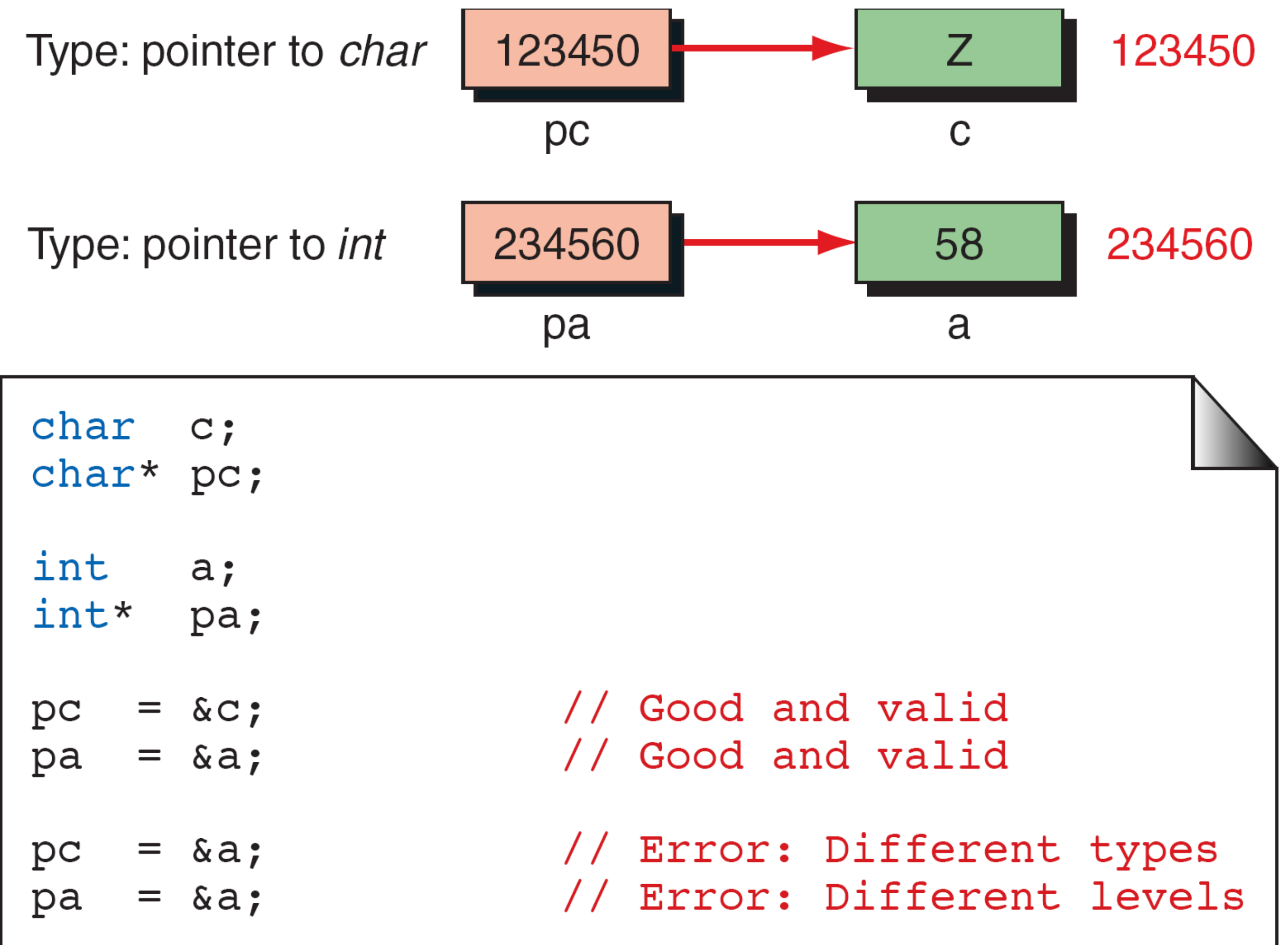
PROGRAM 9-7 Demonstrate Size of Pointers

```
19      int   sizeofpa      = sizeof(pa);
20      int   sizeofStarpa = sizeof(*pa);
21
22      double  x;
23      double* px;
24      int sizeofx       = sizeof(x);
25      int sizeofpx      = sizeof(px);
26      int sizeofStarpx = sizeof(*px);
27
28   // Statements
29      printf("sizeof(c): %3d | ",  sizeofc);
30      printf("sizeof(pc): %3d | ", sizeofpc);
31      printf("sizeof(*pc): %3d\n", sizeofStarpc);
32
33      printf("sizeof(a): %3d | ",  sizeofa);
34      printf("sizeof(pa): %3d | ", sizeofpa);
35      printf("sizeof(*pa): %3d\n", sizeofStarpa);
36
```

PROGRAM 9-7 Demonstrate Size of Pointers

```
37    printf("sizeof(x): %3d | ",  sizeofx);
38    printf("sizeof(px): %3d | ", sizeofpx);
39    printf("sizeof(*px): %3d\n", sizeofStarpx);
40
41    return 0;
42 } // main
```

```
Results:
sizeof(c):   1 | sizeof(pc):   4 | sizeof(*pc):   1
sizeof(a):   4 | sizeof(pa):   4 | sizeof(*pa):   4
sizeof(x):   8 | sizeof(px):   4 | sizeof(*px):   8
```




```
char  c;
char* pc;

int   a;
int*  pa;

pc  = &c;              // Good and valid
pa  = &a;              // Good and valid

pc  = &a;              // Error: Different types
pa  = &a;              // Error: Different levels
```

FIGURE 9-22 Dereference Type Compatibility

Pointer to Void

- A generic type that is not associated with a reference type
- Compatible for assignment purpose only with all other pointer types
  - Any pointer can be assigned to a void pointer
  - A void pointer can be assigned to any pointer
- NULL pointer?

Note

A void pointer cannot be dereferenced.

Casting Pointers

- pc = (char *) & a;

- // Local declarations
```
    void* pVoid;
    char* pChar;
    int* pInt;
// Statements
    pVoid = pChar;
    pInt = pVoid;
    pInt = (int *) pChar;
```

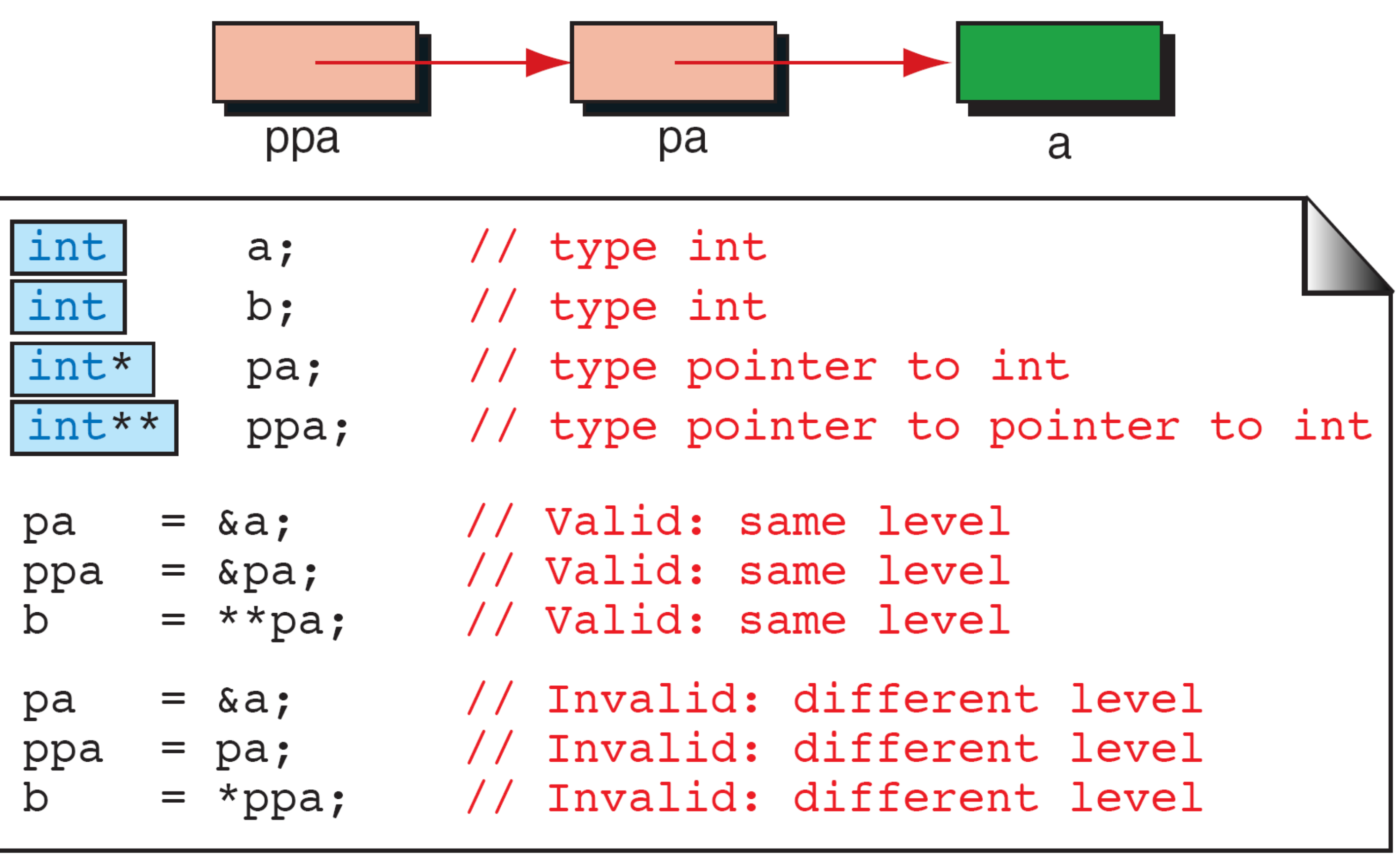


FIGURE 9-23 Dereference Level Compatibility

9-5 Lvalue and Rvalue

In C, an expression is either an lvalue or an rvalue. As you know, every expression has a value. But the value in an expression (after evaluation) can be used in two different ways.

Topics discussed in this section:

Pointer Examples

	Expression Type[a]	Comments
1.	identifier	Variable identifier
2.	expression[...]	Array indexing
3.	(expression)	Expression must already be lvalue
4.	*expression	Dereferenced expression
5.	expression.name	Structure selection
6.	expression->name	Structure indirect selection
7.	function call	If function uses *return* by address

[a] Expression types 5, 6, and 7 have not yet been covered.

Table 9-1 ***lvalue* Expressions**

Note

The right operand of an assignment operator must be an *rvalue* expression.

Type of Expression	Examples	
Address operator	`&score`	
Postfix increment/decrement	`x++`	`y--`
Prefix increment/decrement	`++x`	`--y`
Assignment (left operand)	`x = 1`	`y += 3`

Table 9-2 Operators That Require *lvalue* Expressions

Expression	Problem
`a + 2 = 6;`	a + 2 is an rvalue and cannot be the left operand in an assignment; it is a temporary value that does not have an address; no place to store 6.
`&(a + 2);`	a + 2 is an rvalue, and the address operator needs an lvalue; rvalues are temporary values and do not have addresses.
`&4;`	Same as above (4 is an rvalue).
`(a + 2)++;` `++(a + 2);`	Postfix and prefix operators require lvalues; (a + 2) is an rvalue.

Table 9-3 **Invalid *rvalue* Expressions**

PROGRAM 9-8 Convert Seconds to Hours, Minutes, and Seconds

```
1   /* ==================== secToHours ====================
2      Given time in seconds, convert it to hours, minutes,
3      and seconds.
4         Pre    time in seconds
5                addresses of hours, minutes, seconds
6         Post   hours, minutes, seconds calculated
7         Return error indicator--1 success, 0 bad time
8   */
9   int secToHours (long time,
10                  int* hours, int* minutes, int* seconds)
11  {
12  // Local Declarations
13     long localTime;
14
15  // Statements
16     localTime = time;
17     *seconds  = localTime % 60;
18     localTime = localTime / 60;
```

PROGRAM 9-8 Convert Seconds to Hours, Minutes, and Seconds

```
19
20     *minutes  = localTime % 60;
21
22     *hours    = localTime / 60;
23
24     if (*hours > 24)
25        return 0;
26     else
27        return 1;
28  }  // secToHours
```

Note

Create local variables when a value parameter will be changed within a function so that the original value will always be available for processing.

Note

When several values need to be sent back to the calling function, use address parameters for all of them. Do not return one value and use address Parameters for the others.

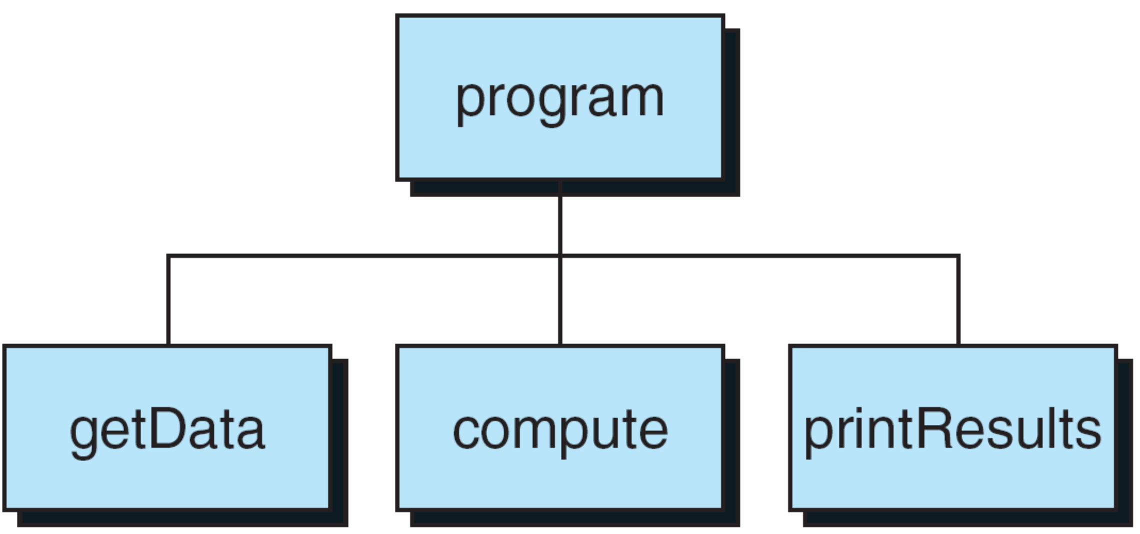


FIGURE 9-24 A Common Program Design

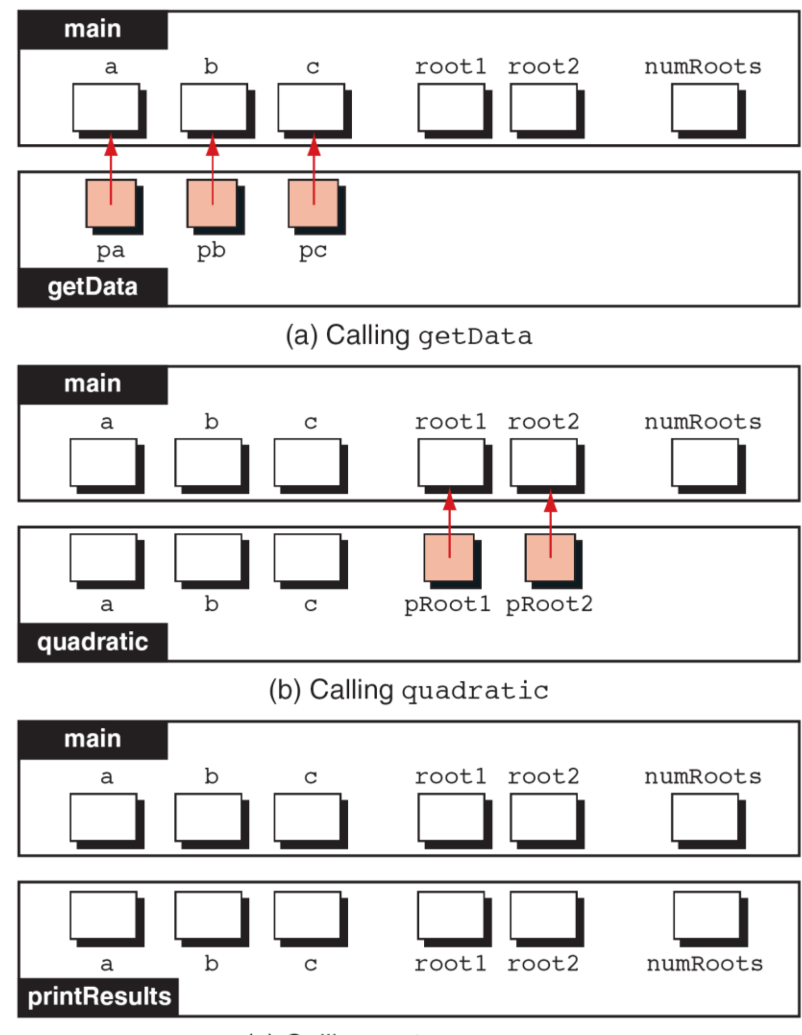


FIGURE 9-25 Using Pointers as Parameters

PROGRAM 9-9 Quadratic Roots

```
1   /* Test driver for quadratic function.
2         Written by:
3         Date:
4   */
5   #include <stdio.h>
6   #include <math.h>
7
8   // Function Declarations
9   void getData      (int*    pa,      int* pb, int* pc);
10  int  quadratic    (int     a,       int  b,  int  c,
11                     double* pRoot1, double* pRoot2);
12  void printResults (int     numRoots,
13                     int     a,      int  b, int  c,
14                     double  root1, double  root2);
15
16  int main (void)
17  {
```

PROGRAM 9-9 Quadratic Roots

```
18  // Local Declarations
19     int    a;
20     int    b;
21     int    c;
22     int    numRoots;
23     double root1;
24     double root2;
25     char   again = 'Y';
26
27  // Statements
28     printf("Solve quadratic equations.\n\n");
29     while (again == 'Y' || again == 'y')
30        {
31         getData (&a, &b, &c);
32         numRoots = quadratic (a, b, c, &root1, &root2);
33         printResults (numRoots, a, b, c, root1, root2);
34
35        printf("\nDo you have another equation (Y/N): ");
36         scanf (" %c", &again);
37        } // while
```

PROGRAM 9-9 Quadratic Roots

```
38      printf("\nThank you.\n");
39      return 0;
40   }  // main
41
42   /* ================== getData ==================
43      Read coefficients for quadratic equation.
44         Pre   a, b, and c contains addresses
45         Post  data read into addresses in main
46   */
47   void getData   (int* pa, int* pb, int* pc)
48   {
49   // Statements
50      printf("Please enter coefficients a, b, & c: ");
51      scanf ("%d%d%d", pa, pb, pc);
52
53      return;
54   }  // getData
55
56   /* ================== quadratic ==================
```

PROGRAM 9-9 Quadratic Roots

```
57      Compute the roots for a quadratic equation.
58         Pre     a, b, & c are the coefficients
59                 pRoot1 & pRoot2 are variable pointers
60         Post    roots computed, stored in calling function
61         Return  2 two roots,
62                  1 one root,
63                  0 imaginary roots
64                 -1 not quadratic coefficients.
65   */
66   int quadratic (int a, int b, int c,
67                  double* pRoot1, double* pRoot2)
68   {
69   // Local Declarations
70      int result;
71   
72      double discriminant;
73      double root;
74   
75   // Statements
76      if (a == 0 && b == 0)
77         result = -1;
```

PROGRAM 9-9 Quadratic Roots

```
78      else
79         if (a == 0)
80            {
81             *pRoot1 = -c / (double) b;
82             result = 1;
83            } // a == 0
84         else
85            {
86             discriminant =  b * b  - (4 * a * c);
87             if (discriminant >= 0)
88                {
89                 root    = sqrt(discriminant);
90                 *pRoot1 = (-b + root) / (2 * a);
91                 *pRoot2 = (-b - root) / (2 * a);
92                 result  = 2;
93                } // if >= 0
94             else
95                 result = 0;
96            } // else
97      return result;
98   }  // quadratic
99
100  /* ================= printResults ================
```

PROGRAM 9-9 **Quadratic Roots**

```
101      Prints the factors for the quadratic equation.
102         Pre    numRoots contains 0, 1, 2
103                a, b, c contains original coefficients
104                root1 and root2 contains roots
105         Post   roots have been printed
106   */
107   void printResults  (int    numRoots,
108                       int    a,  int b,  int c,
109                       double root1, double root2)
110   {
111   // Statements
112      printf("Your equation: %dx**2 + %dx + %d\n",
113              a, b, c);
114      switch (numRoots)
115         {
116          case 2:  printf("Roots are: %6.3f & %6.3f\n",
117                           root1, root2);
118                   break;
```

PROGRAM 9-9 Quadratic Roots

```
119             case 1:  printf("Only one root: %6.3f\n",
120                              root1);
121                      break;
122             case 0:  printf("Roots are imaginary.\n");
123                      break;
124             default: printf("Invalid coefficients\n");
125                      break;
126            } // switch
127        return;
128     }  // printResults
129     // ================== End of Program =================
```

```
Results:
Solve quadratic equations.

Please enter the coefficients a, b, & c: 2 4 2
Your equation: 2x**2 + 4x + 2
Roots are: -1.000 & -1.000
```

PROGRAM 9-9 Quadratic Roots

```
Do you have another equation (Y/N): y
Please enter the coefficients a, b, & c: 0 4 2
Your equation: 0x**2 + 4x + 2
Only one root: -0.500

Do you have another equation (Y/N): y
Please enter the coefficients a, b, & c: 2 2 2
Your equation: 2x**2 + 2x + 2
Roots are imaginary.

Do you have another equation (Y/N): y
Please enter the coefficients a, b, & c: 0 0 2
Your equation: 0x**2 + 0x + 2
Invalid coefficients

Do you have another equation (Y/N): y
Please enter coefficients a, b, & c: 1 -5 6
Your equation: 1x**2 + -5x + 6
Roots are:  3.000 &  2.000

Do you have another equation (Y/N): n

Thank you.
```

9-6 Software Engineering

In this chapter, we discuss a general software engineering topic, quality, which can be applied to any topic, including pointers.

Topics discussed in this section:

Quality Defined
Quality Factors
The Quality Circle
Conclusion

Note

Software that satisfies the user's explicit and implicit requirements, is well documented, meets the operating standards of the organization, and runs efficiently on the hardware for which it was developed.

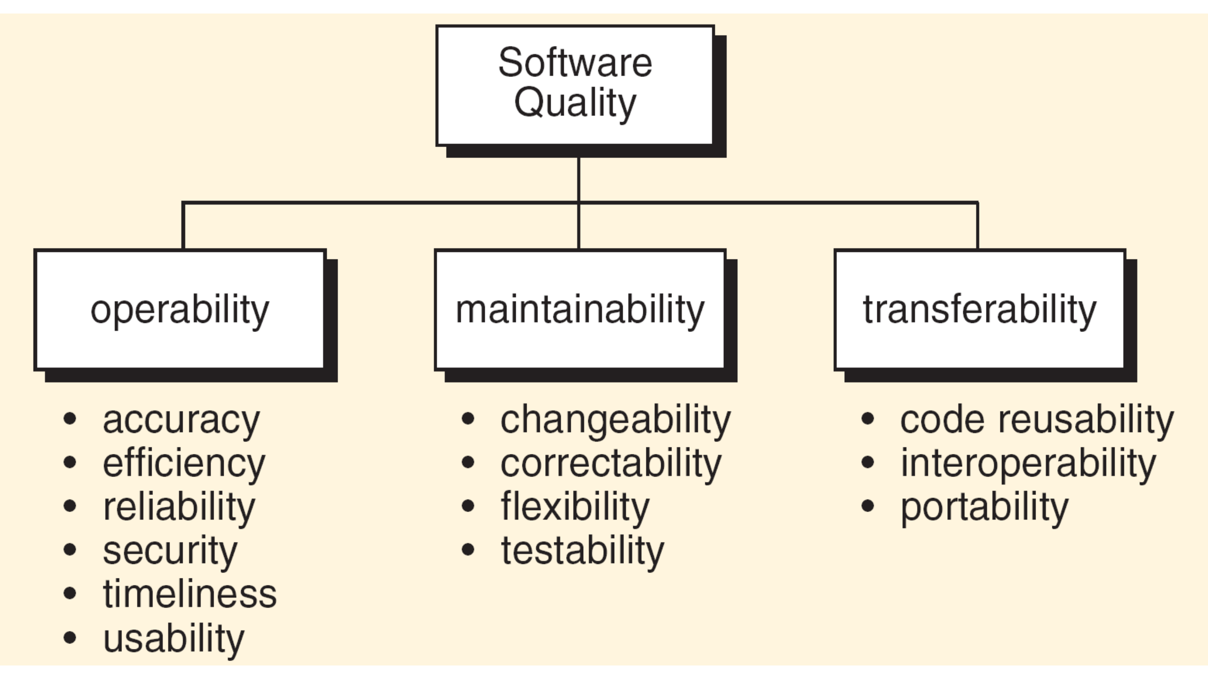


FIGURE 9-26 Software Quality

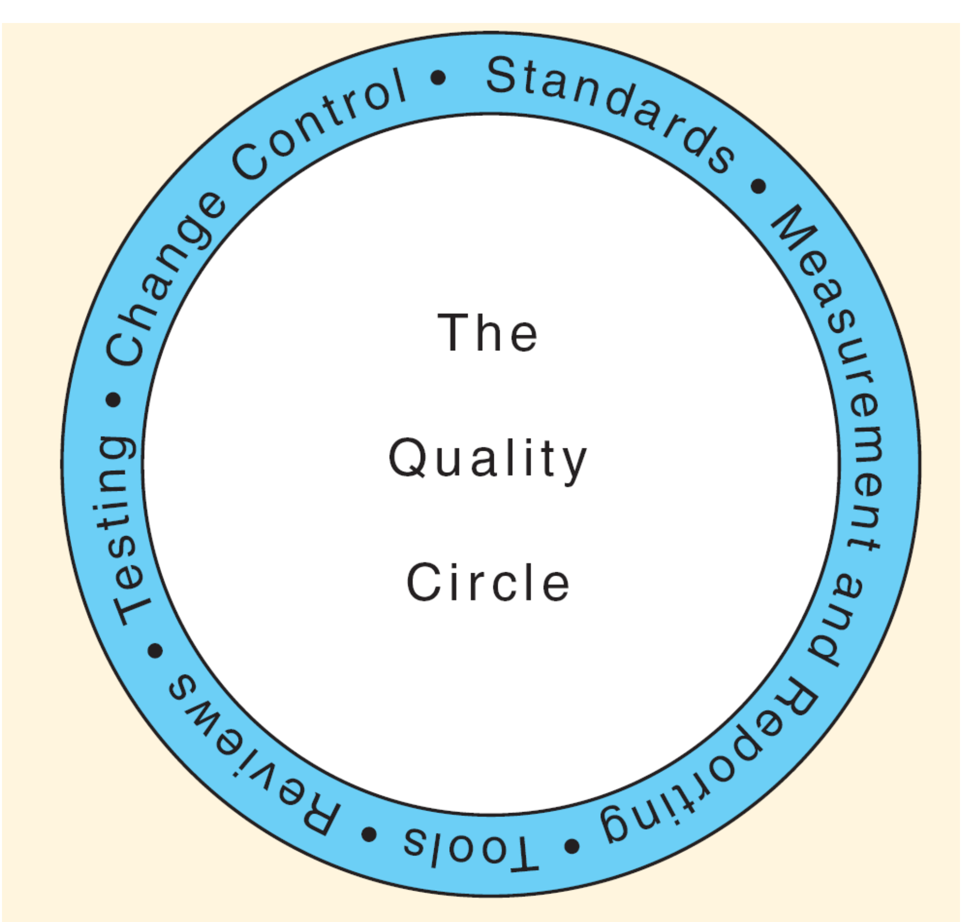


FIGURE 9-27 **The Quality Circle**