Chapter 9

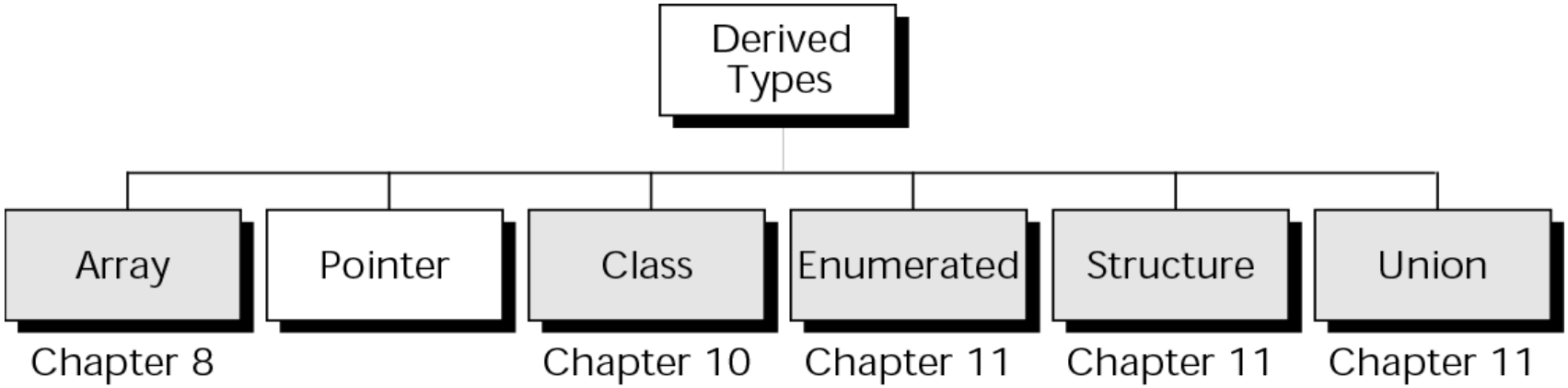
Pointers

OBJECTIVES

After studying this chapter you will be able to:

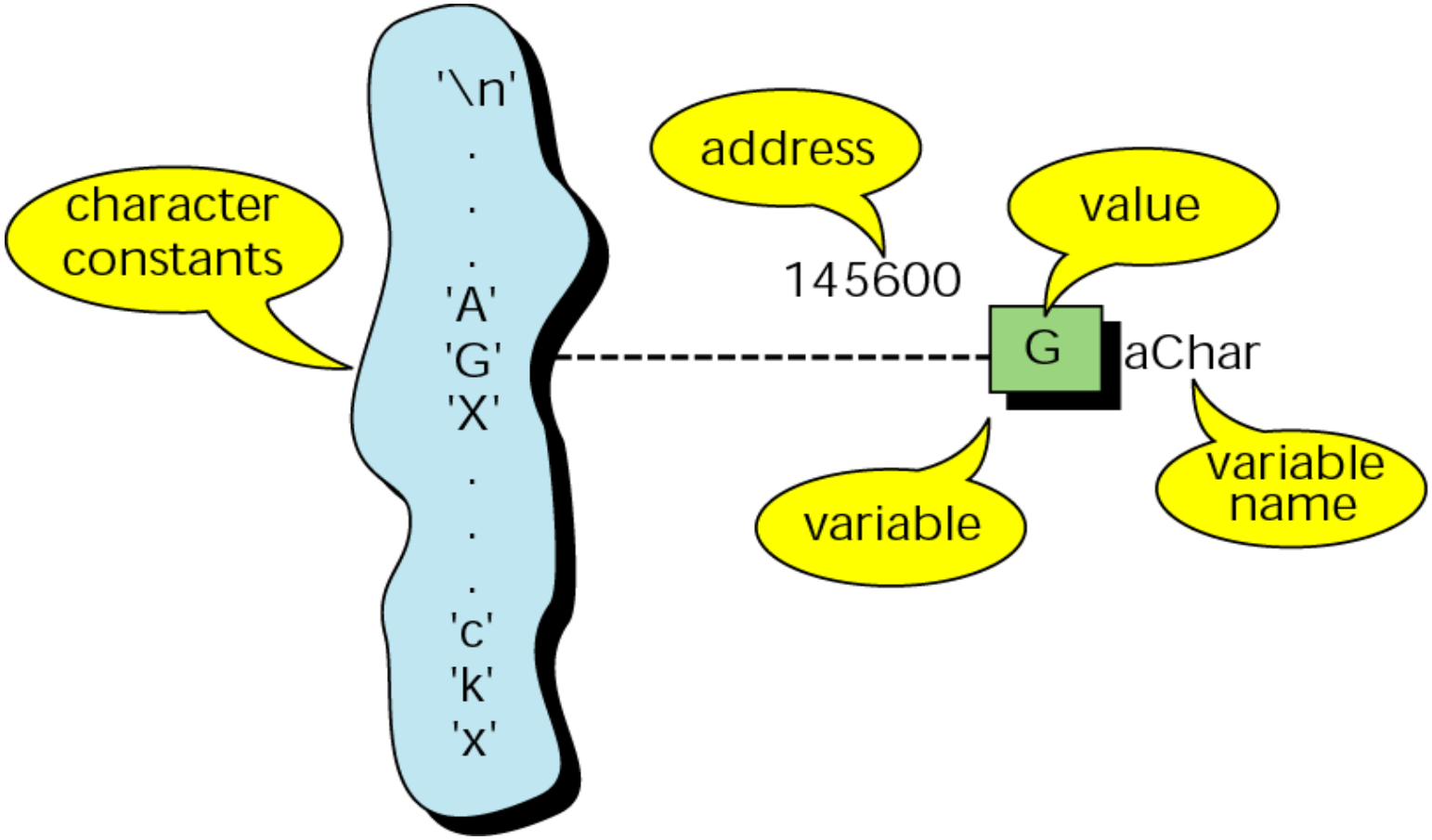
- Understand the design and operation of pointers and use the address and indirection operators.
- Write functions that pass pointers as parameters and that return pointers.
- Use pointers and pointer arithmetic to process the data in an array.
- Use ragged arrays to save space when some rows of an array are not full.
- Describe how memory can be divided between program memory and data memory (global area, heap, and stack).
- Write programs that dynamically allocate memory.

Figure 9-1 Derived types



CONCEPTS

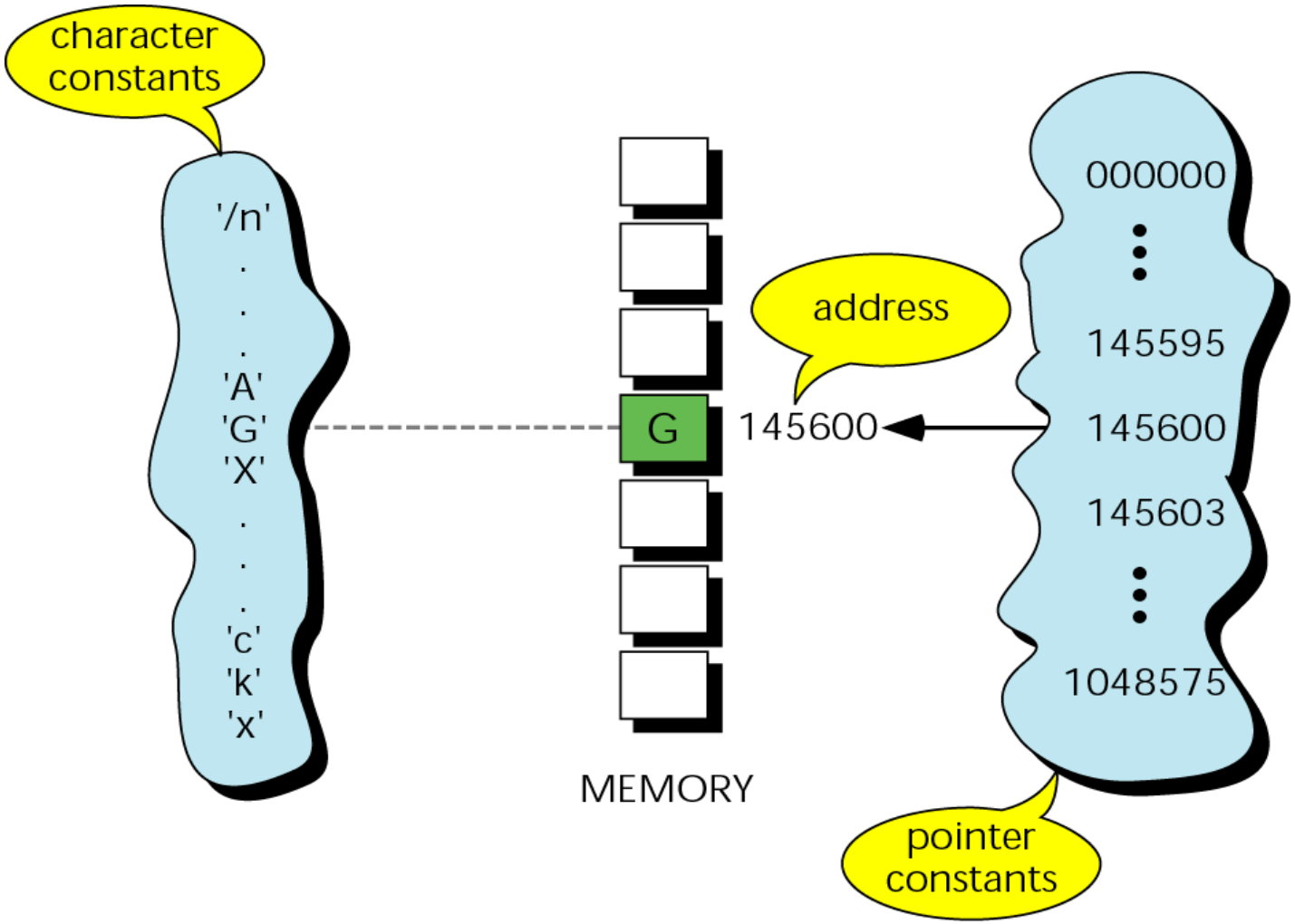
Figure 9-2 Character constants and variables



Note:

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

Figure 9-3 Pointer constants



Note:

When the ampersand (&) is used as a prefix to a variable name, it means "address" of variable. When it is used as a suffix to a type, it means reference parameter.

Figure 9-4 Print character addresses

```
// This program prints character addresses
#include <iostream>
using namespace std;
int  main  ( )
{
   char   a ;
   char   b ;

   cout << &a << &b ;
   return 0 ;
} //  main
```

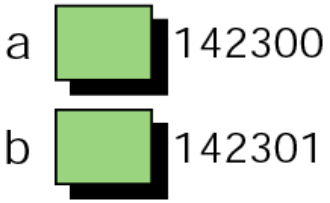
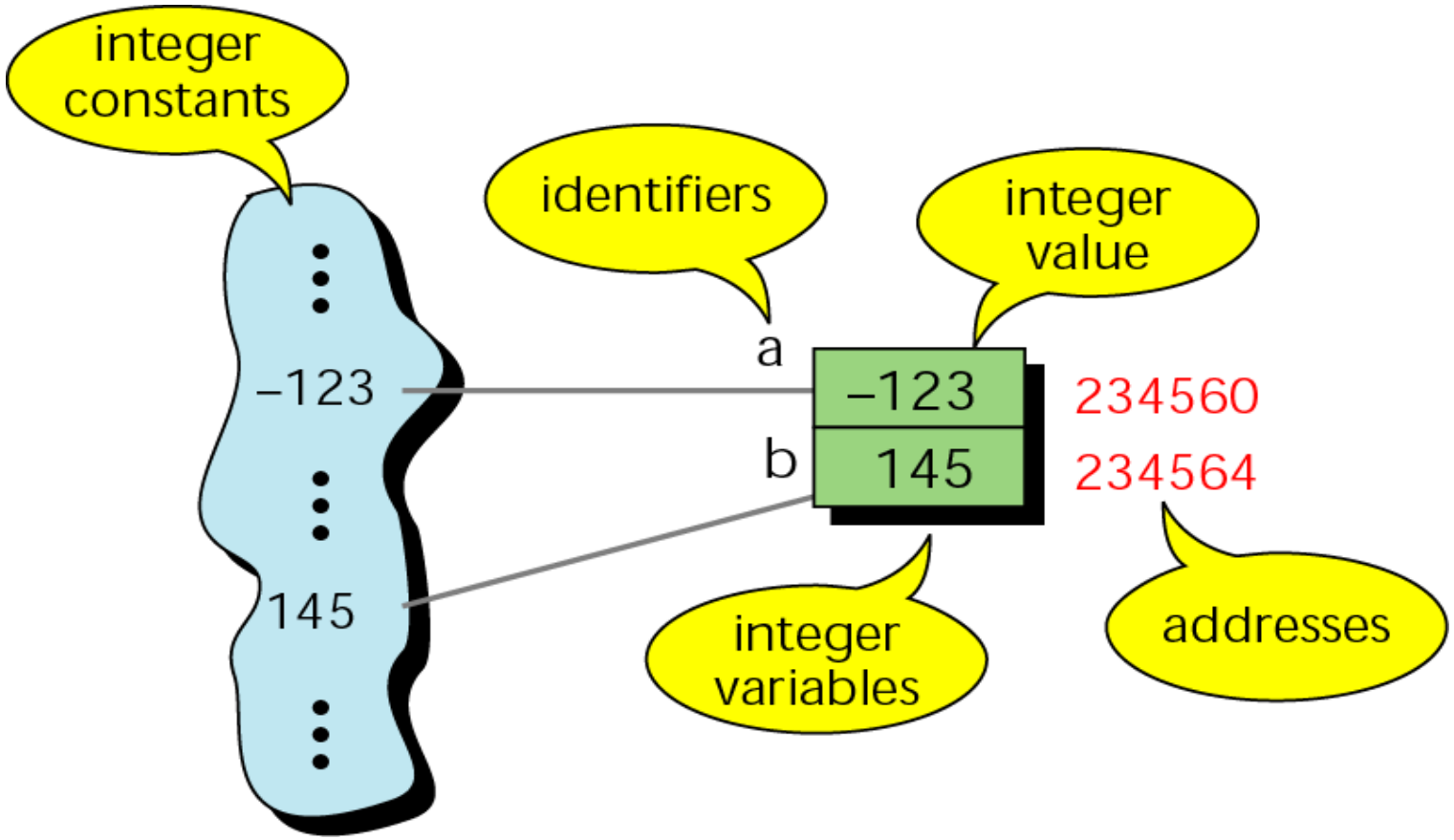


Figure 9-5 Integer constants and variables

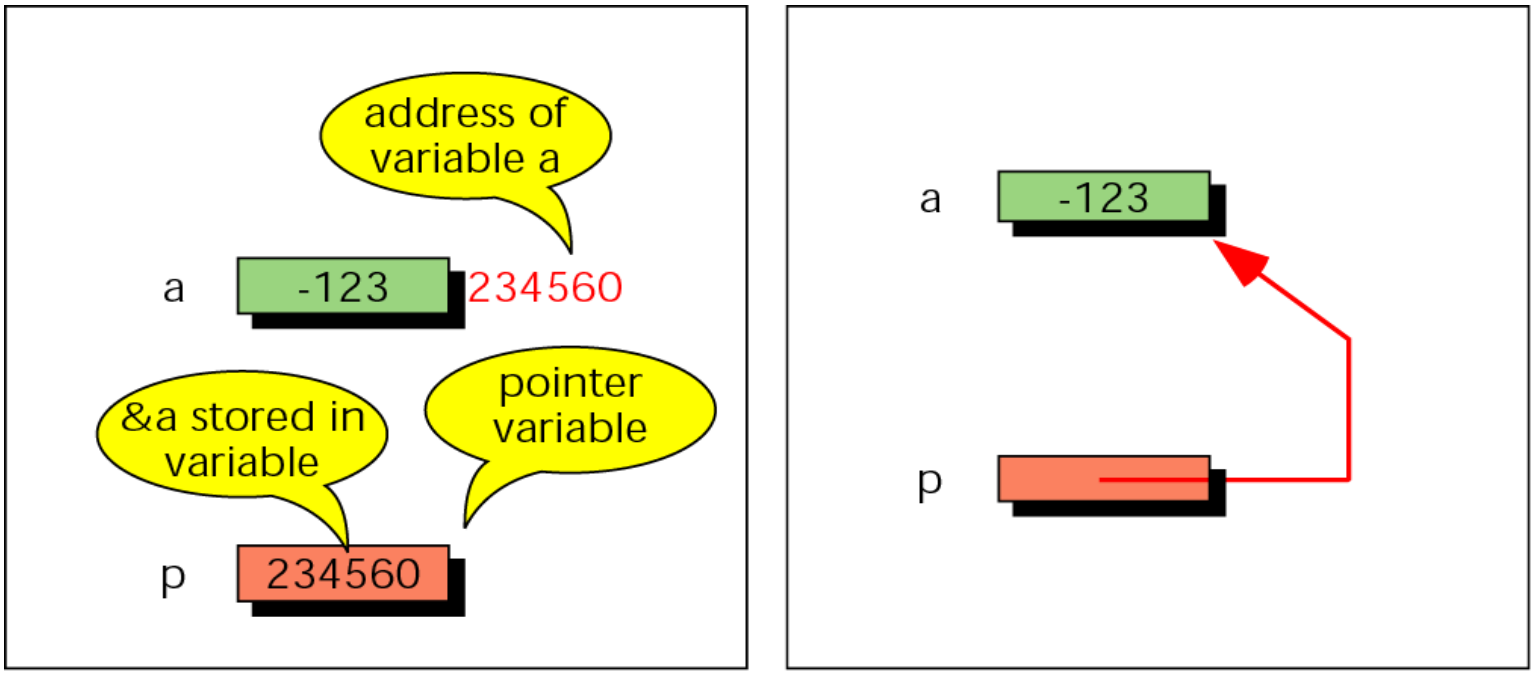


Note:

The address of a variable is the address of the first byte occupied by that variable.

POINTER VARIABLES

Figure 9-6 Pointer variable



Physical representation

Logical representation

Accessing Variables Through Pointers

Figure 9-7 Multiple pointers to a variable

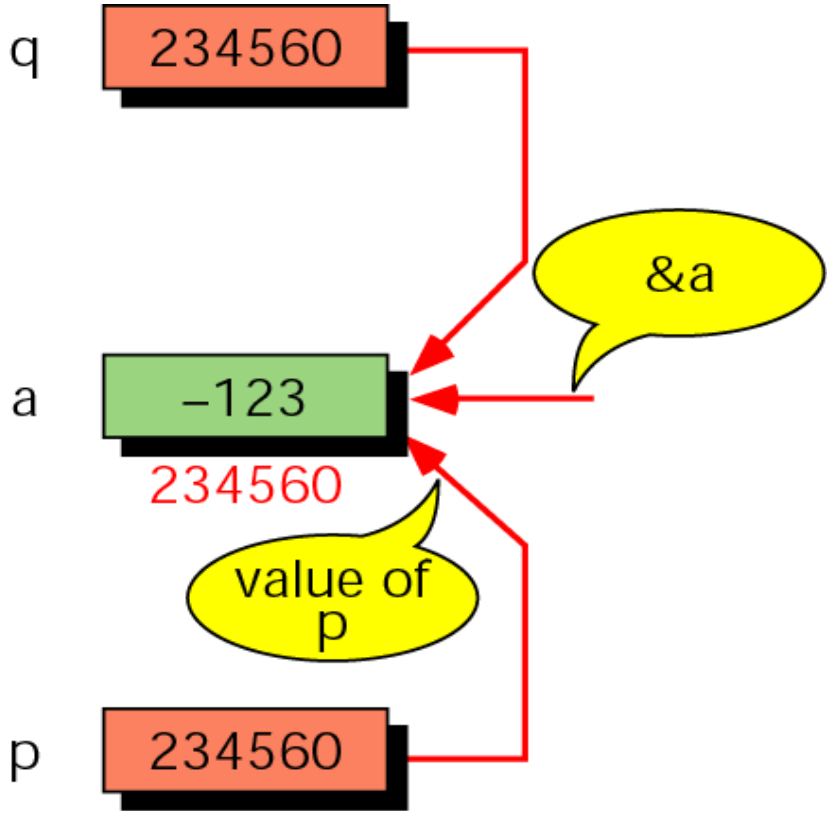


Figure 9-8 Accessing variables through pointers

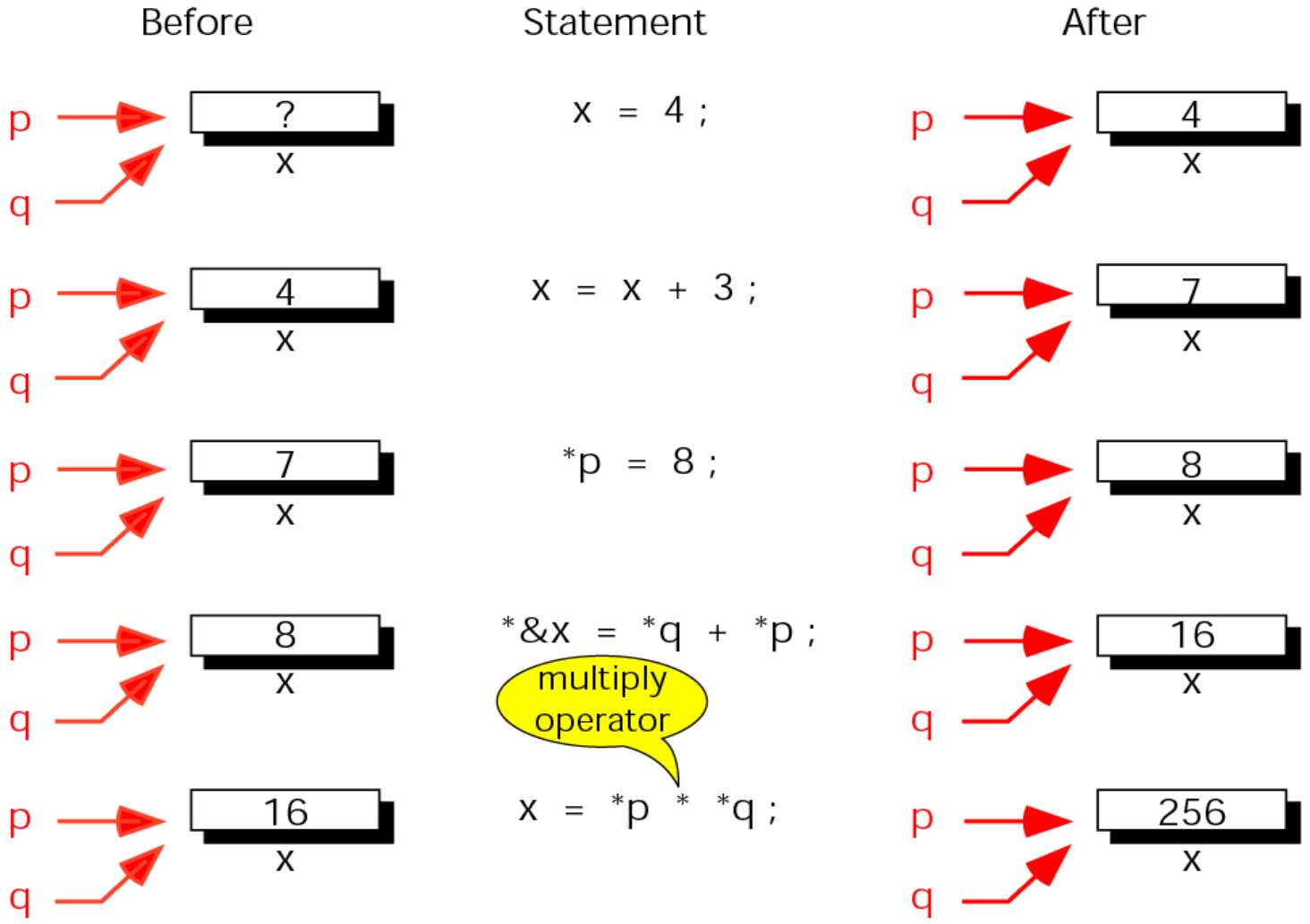
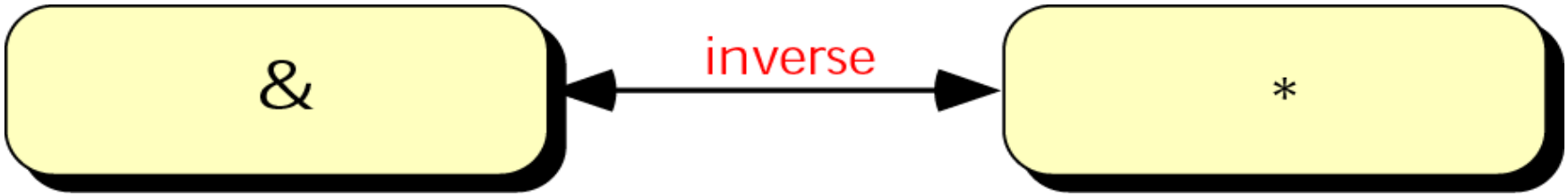


Figure 9-9 Address and indirection operators



POINTER DECLARATION AND DEFINITION

Figure 9-10 Pointer variable declaration

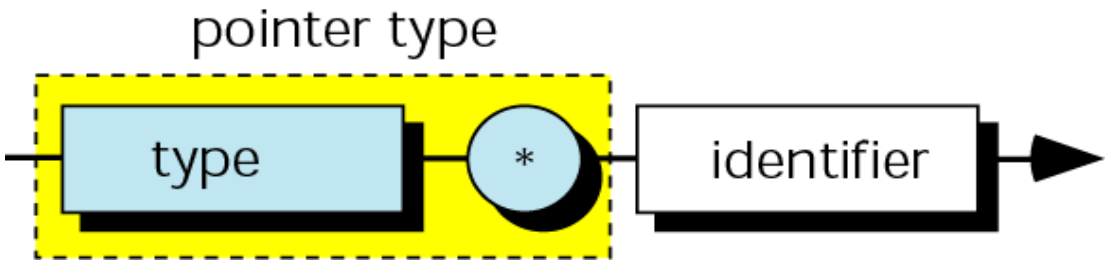
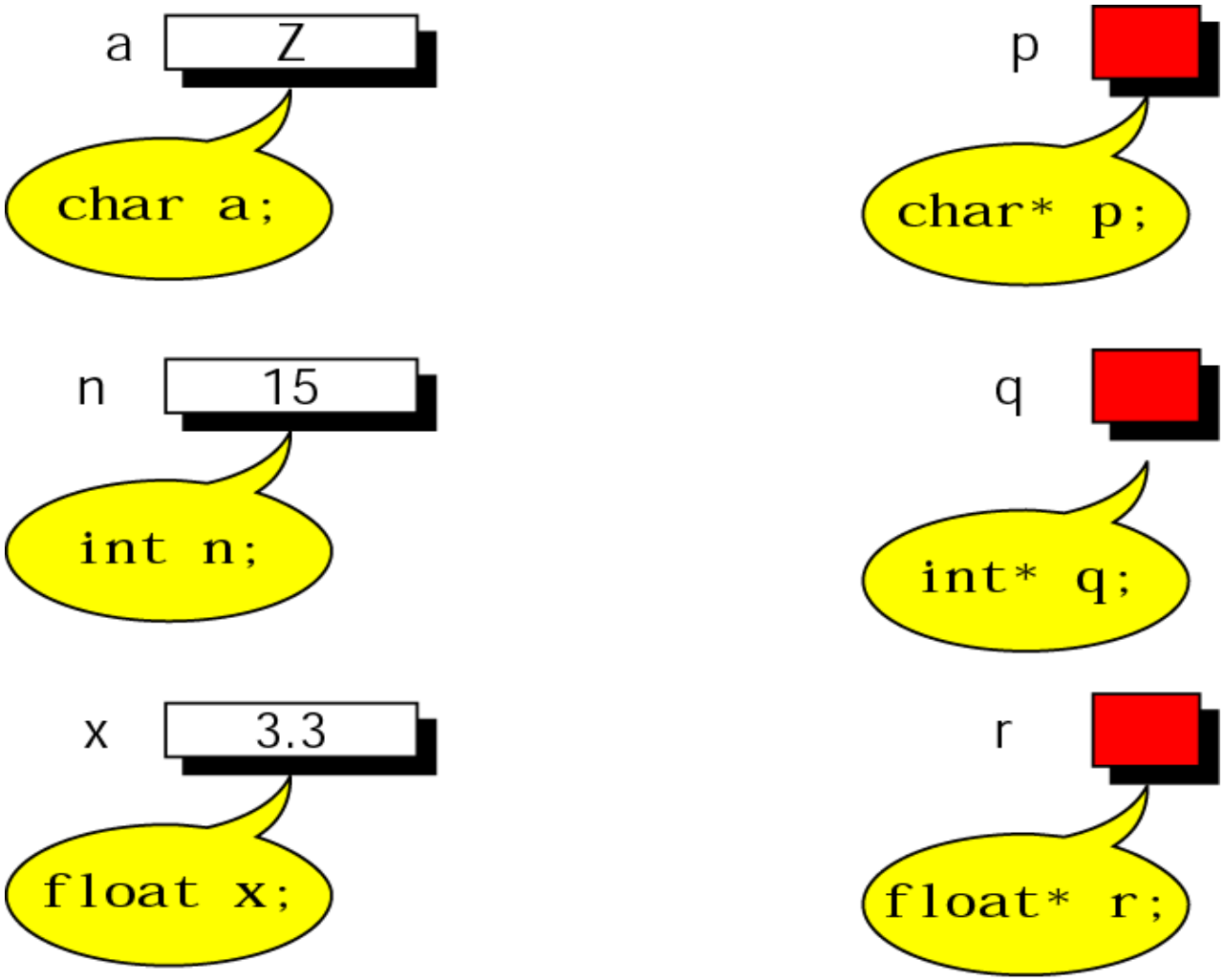


Figure 9-11 Declaring pointer variables



Initialization of Pointer Variables

Figure 9-12 Uninitialized pointers

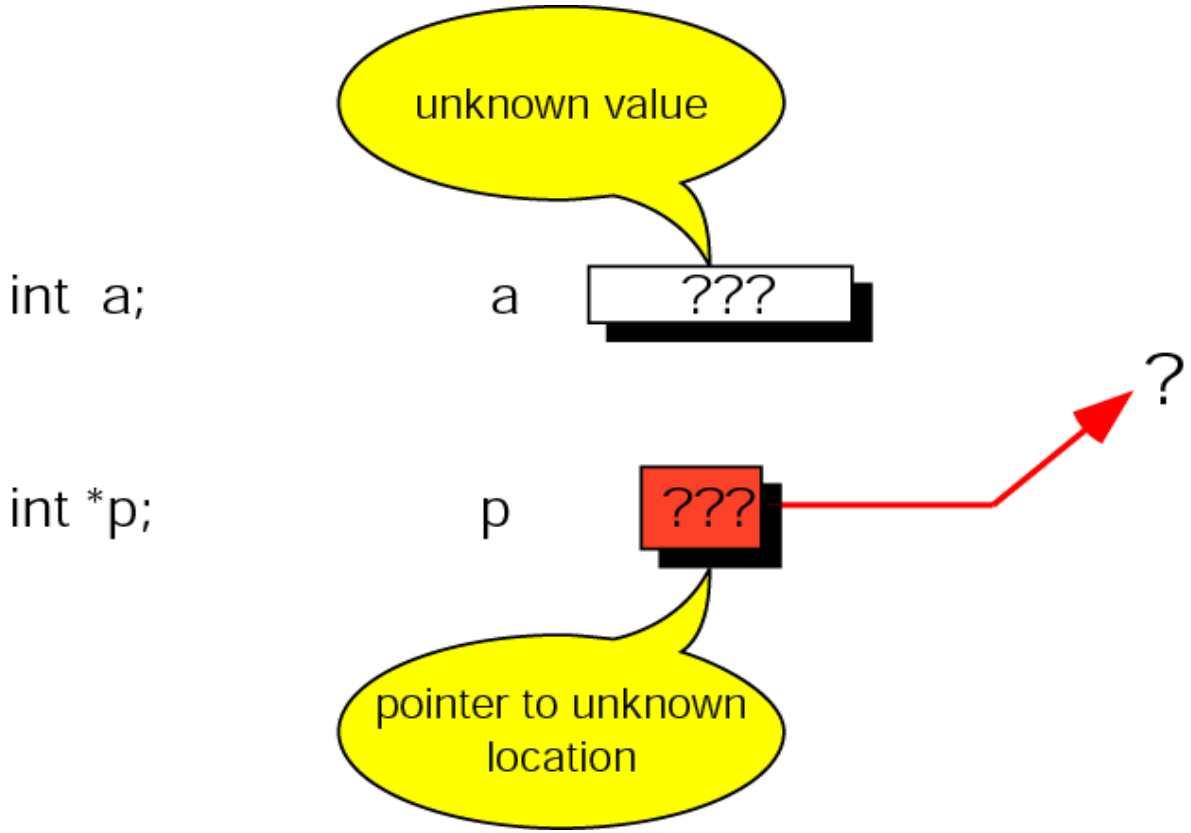


Figure 9-13 Initializing pointer variables

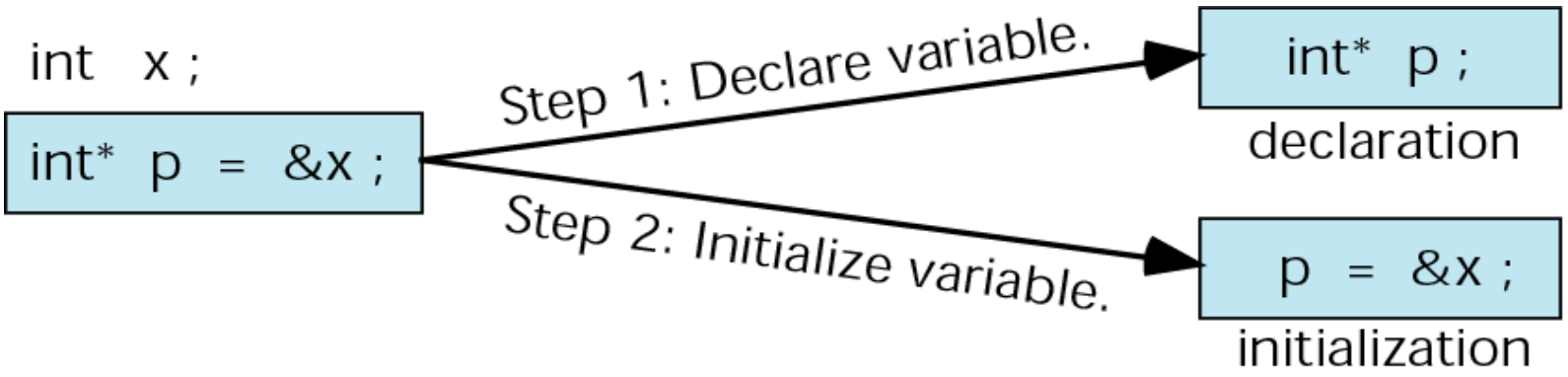


Figure 9-14 Add two numbers using pointers

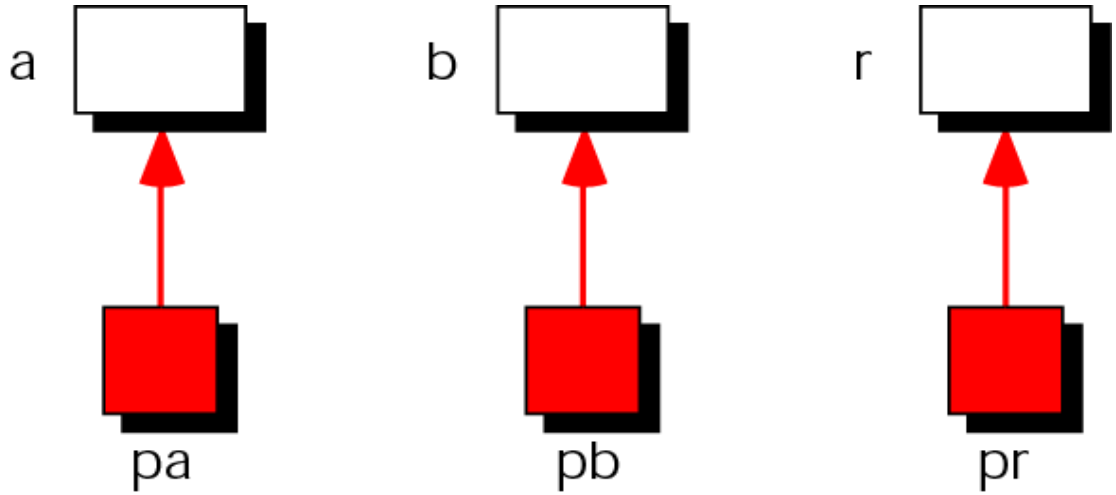


Figure 9-15 Demonstrate pointer flexibility

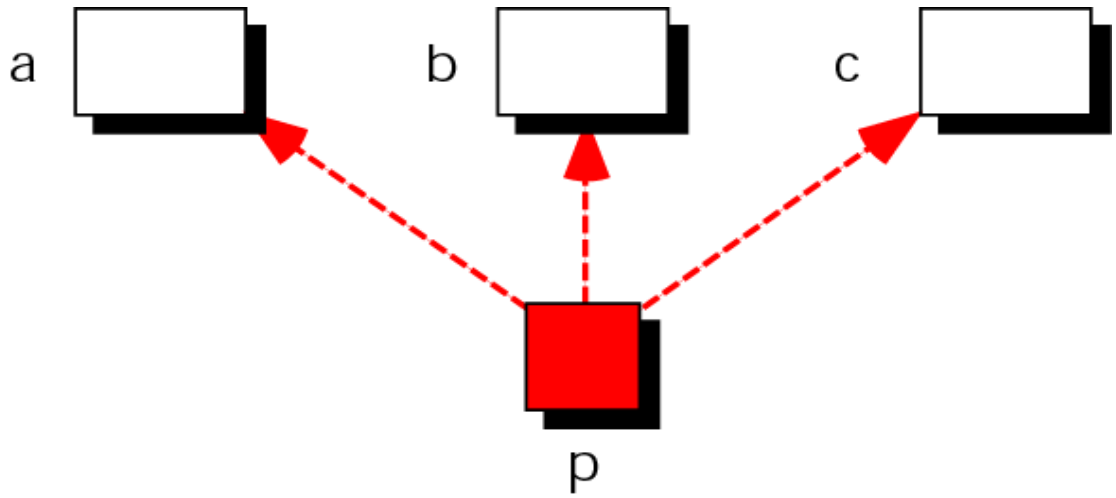
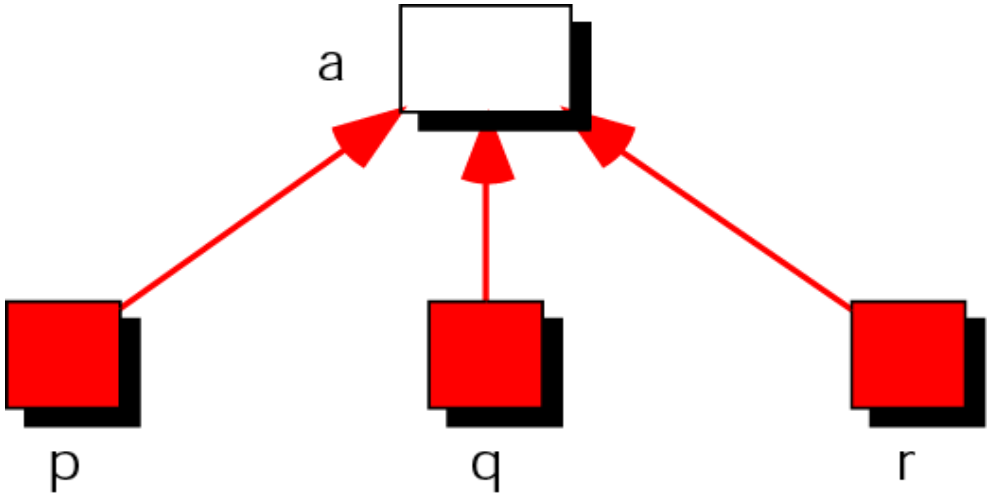


Figure 9-16 Using one variable with many pointers



POINTERS AND FUNCTIONS

Figure 9-17(a) Exchanging values

```
int a = 5;
int b = 7;

// Pass by value
exchange (a, b);
```

```
void exchange (int x, int y)
{
    int temp = x;
    x        = y;
    y        = temp;
    return;
} // exchange
```

(a) Original values unchanged

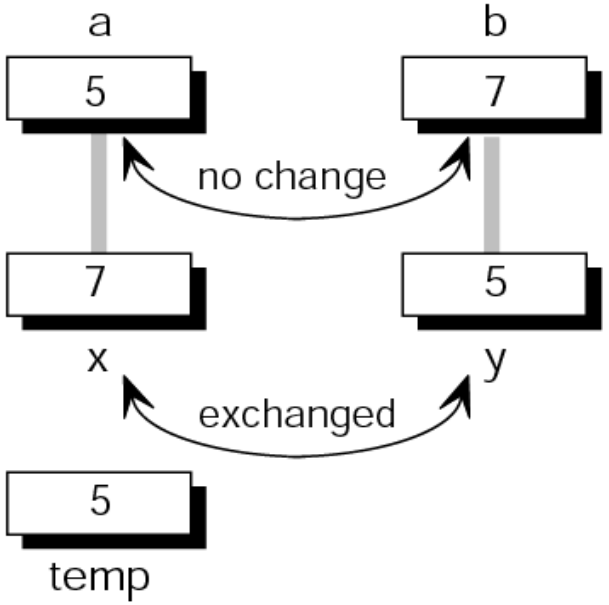


Figure 9-17(b) Exchanging values (continued)

```
int a = 5;
int b = 7;

// Pass by reference
exchange (a, b);
```

```
void exchange (int& x, int& y)
{
   int temp = x;
   x        = y;
   y        = temp;
   return;
} // exchange
```

(b) Original values exchanged

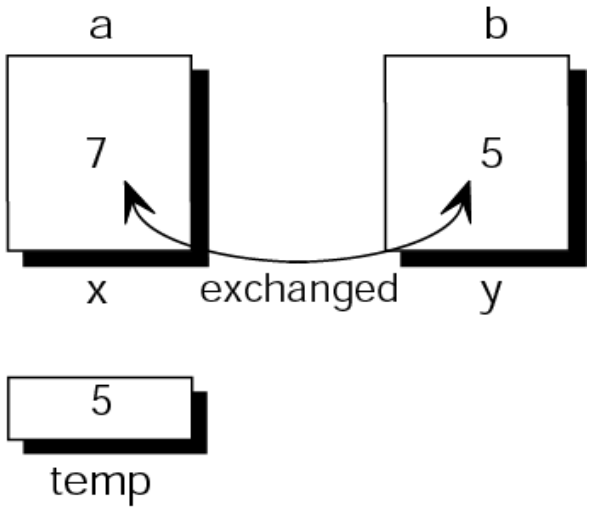


Figure 9-17(c) Exchanging values (continued)

```
int a = 5;
int b = 7;

// Passing pointers
exchange (&a, &b);
```

```
void exchange (int* px, int* py)
{
    int temp    = *px;
    *px         = *py;
    *py         = temp;
    return;
} // exchange
```

(c) Original values exchanged

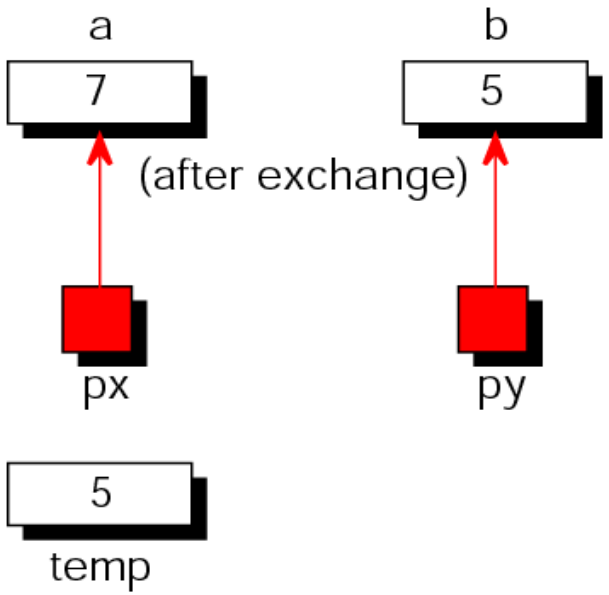
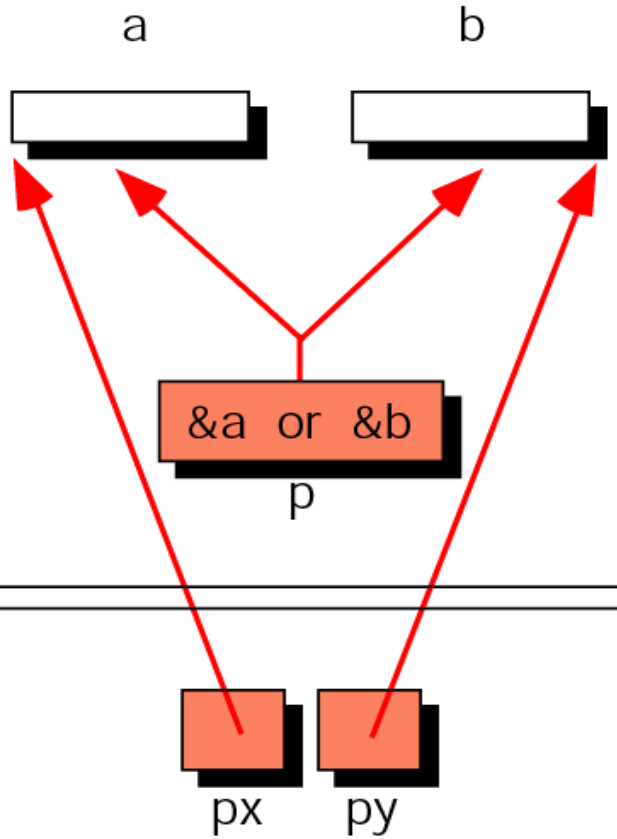


Figure 9-18 Functions returning pointers

```
int*  smaller ( int*, int* ) ;
int  main  ( )
{
   int   a ;
   int   b ;
   int  *p ;
   ...
   cin >> a >> b;
   p = smaller ( &a, &b );
   ...
} // main
```

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



Note:

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

POINTERS TO POINTERS

Figure 9-19 Pointers to pointers

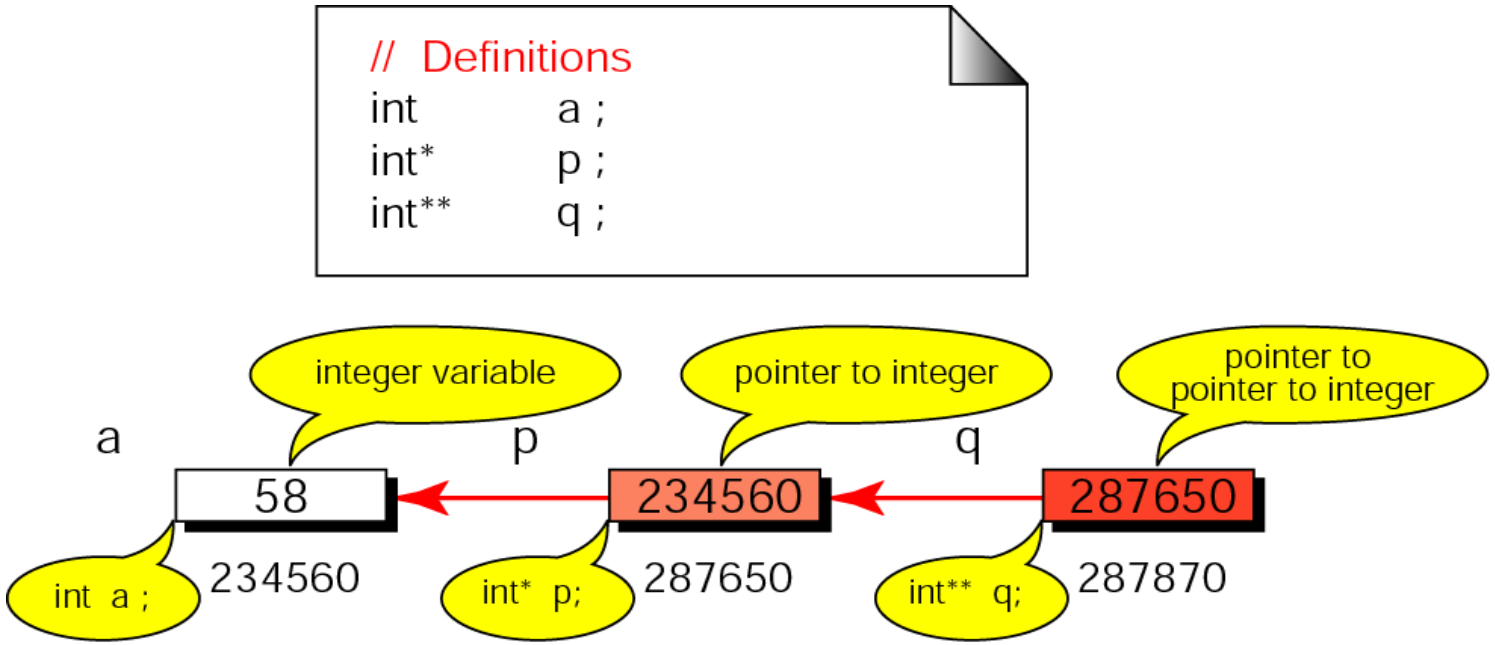
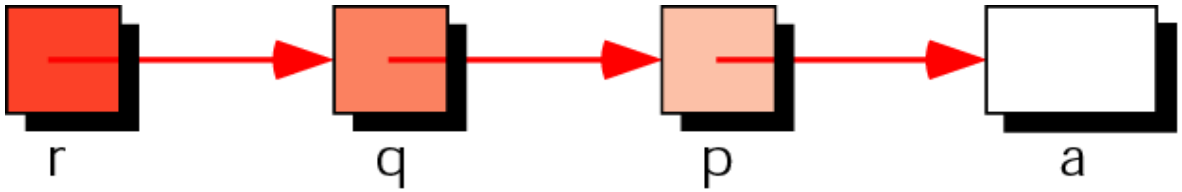
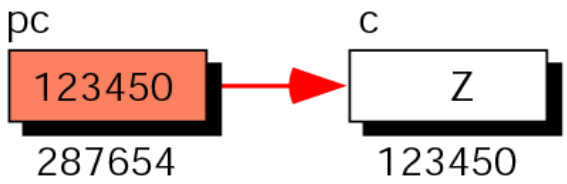


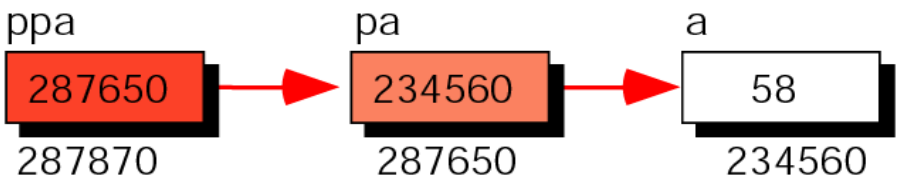
Figure 9-20 Using pointers to pointers



COMPATIBILITY

Figure 9-21 Pointer compatibility





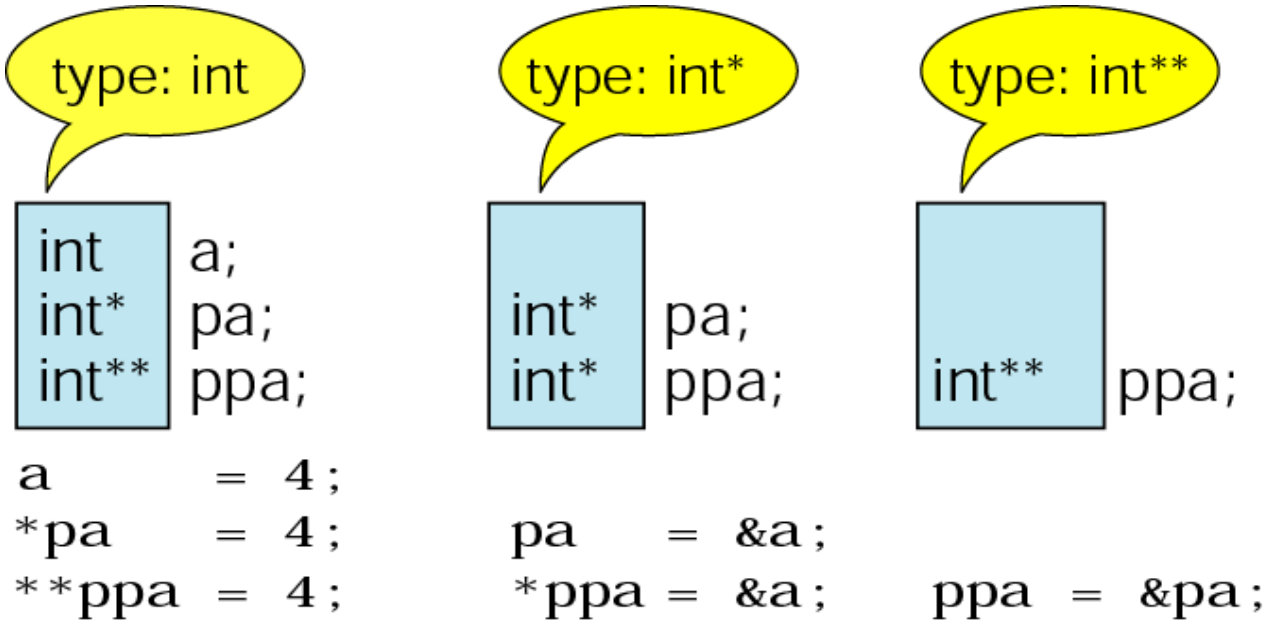

```
char     c = 'z' ;
char*    pc ;

int      a = 58 ;
int*     pa ;
int**    ppa ;

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

// The following are invalid and will generate errors
pc     = &a ;                  // Different types
ppa    = &a ;                  // Different levels
```

Figure 9-22 Pointer types must match



Reading and Writing Pointer Values

LVALUE AND RVALUE

POINTER APPLICATIONS

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 9-23 A common program design

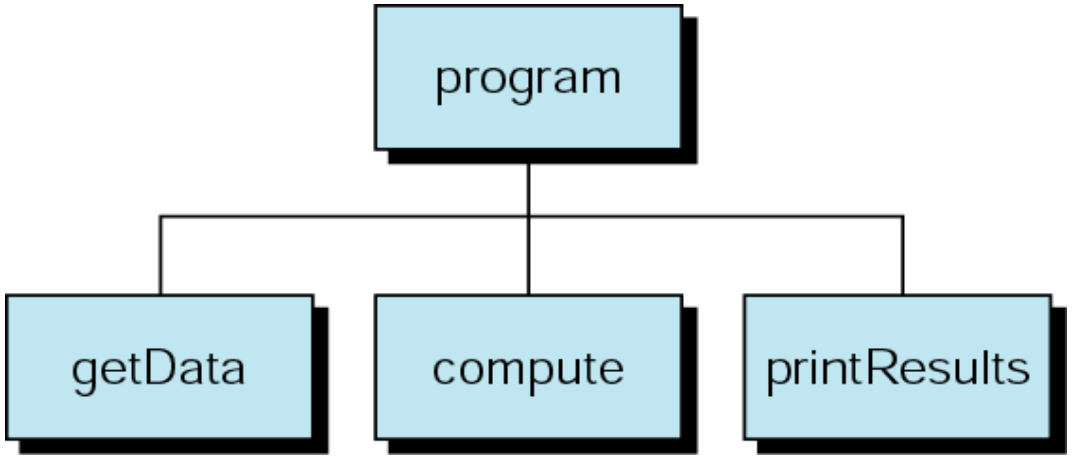
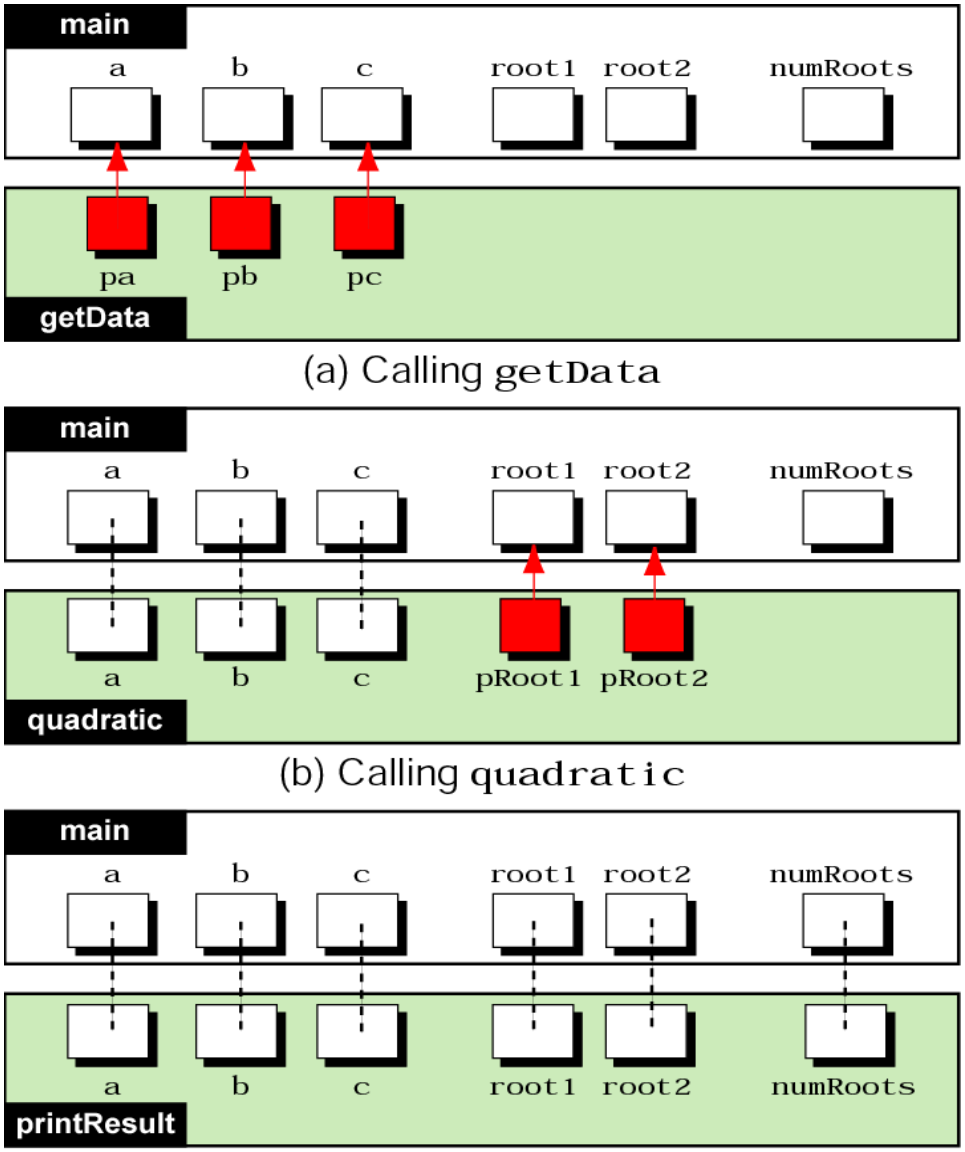


Figure 9-24 Using pointers as parameters



ARRAYS AND POINTERS

Figure 9-25 Pointers to arrays

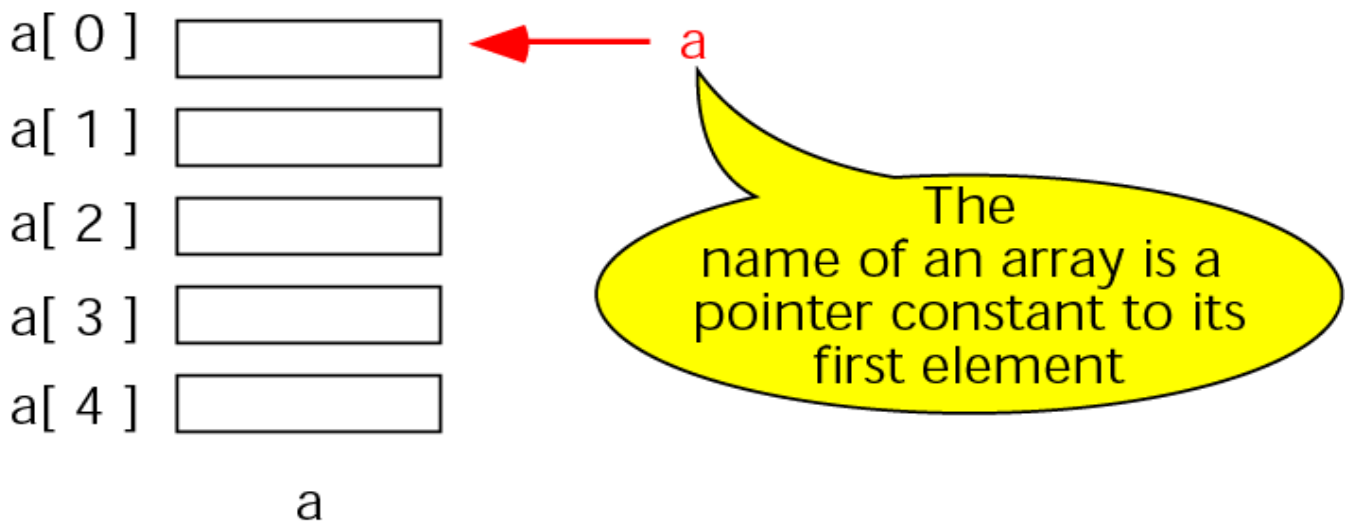


Figure 9-26 Dereference of array name

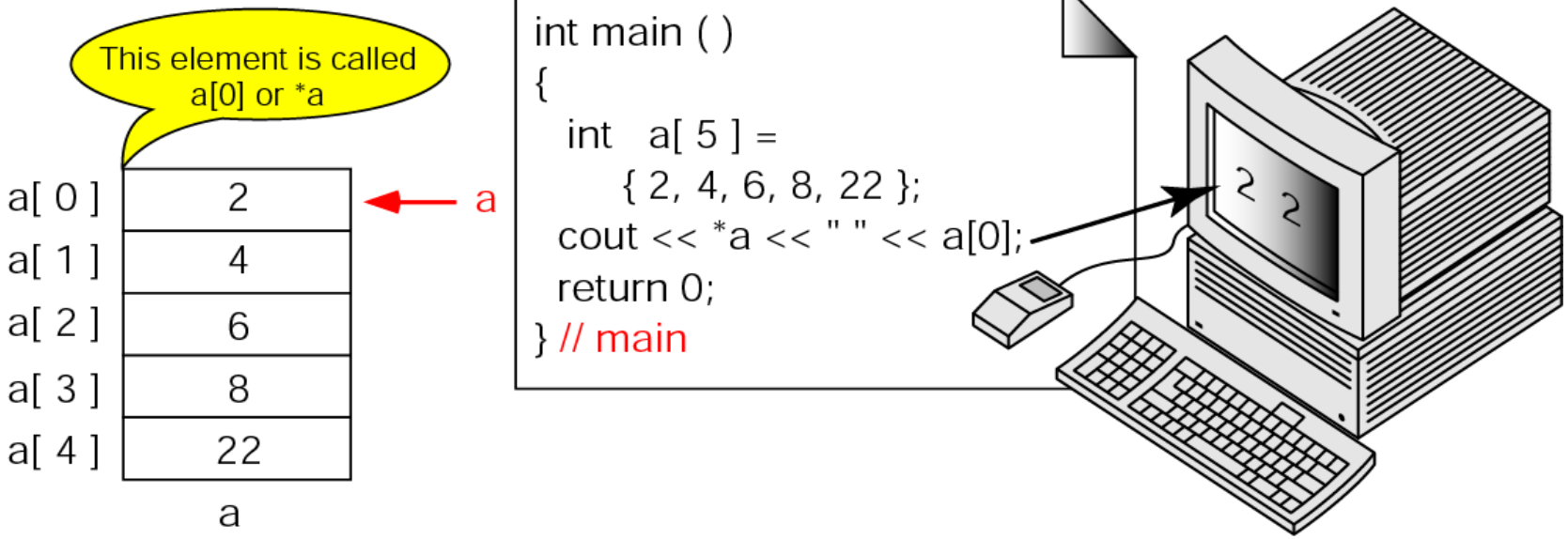
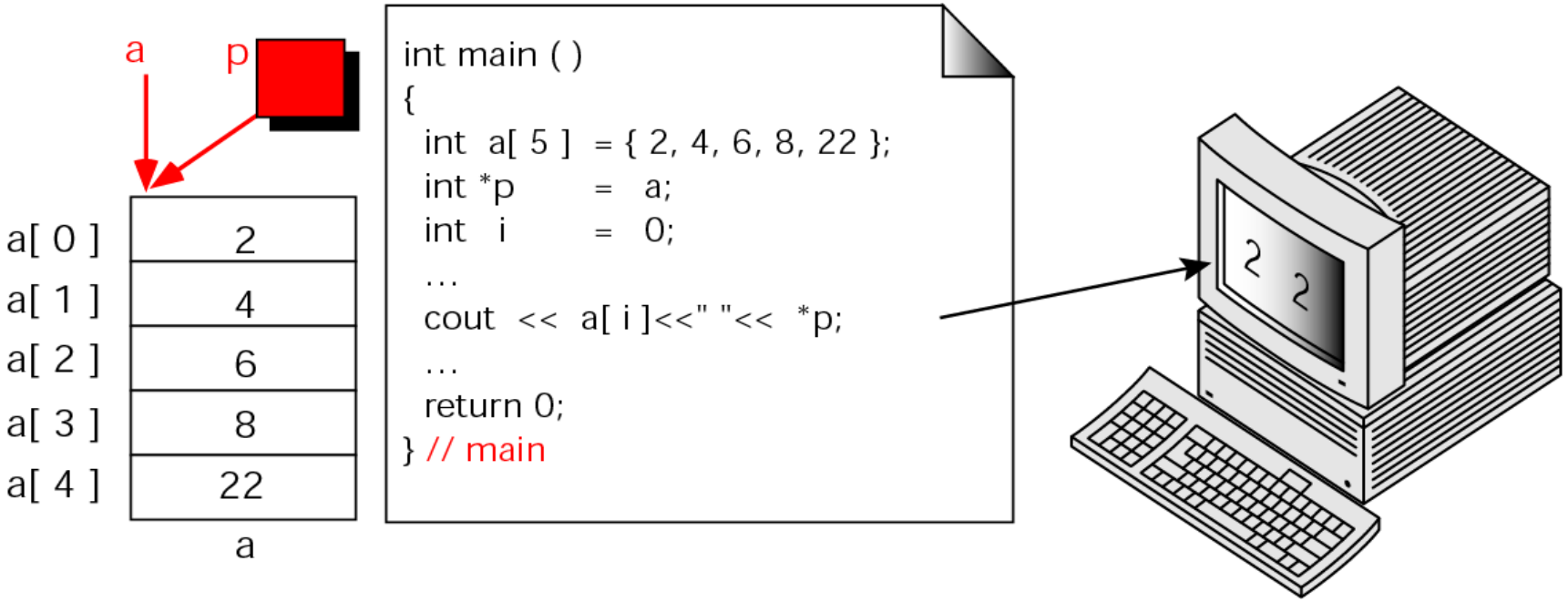


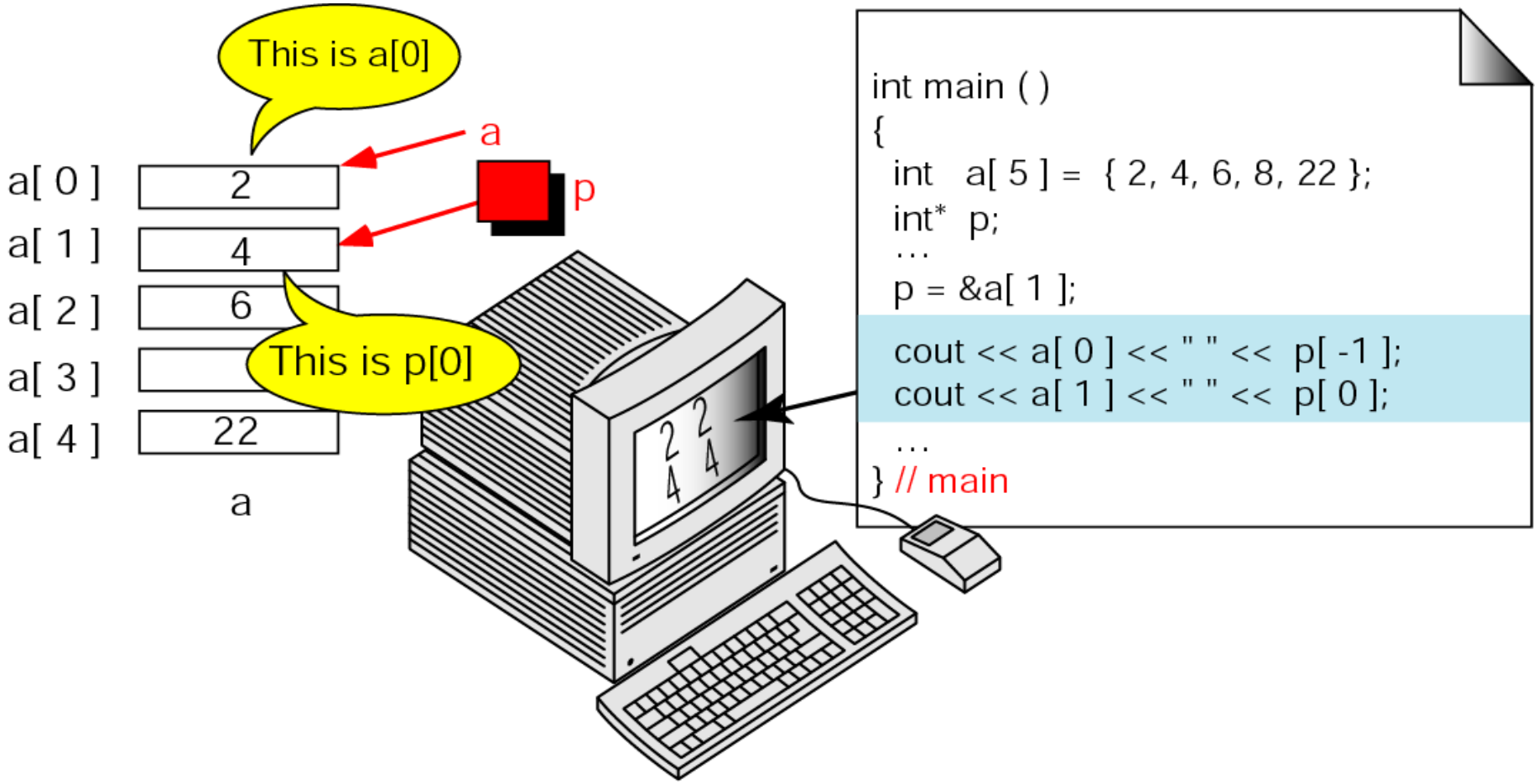
Figure 9-27 Array names as pointers



Note:

To access an array, any pointer to the first element can be used instead of the name of the array.

Figure 9-28 Multiple array pointers



POINTER ARITHMETIC AND ARRAYS

Note:

Given pointer, p, p ± n is a pointer to the value n elements away.

Figure 9-29 Pointer arithmetic

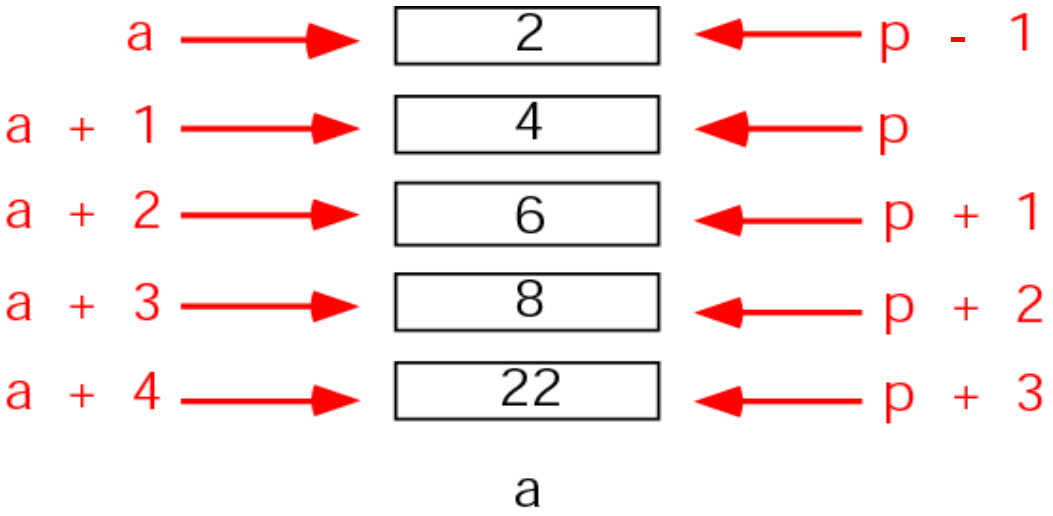


Figure 9-30 Pointer arithmetic and different types

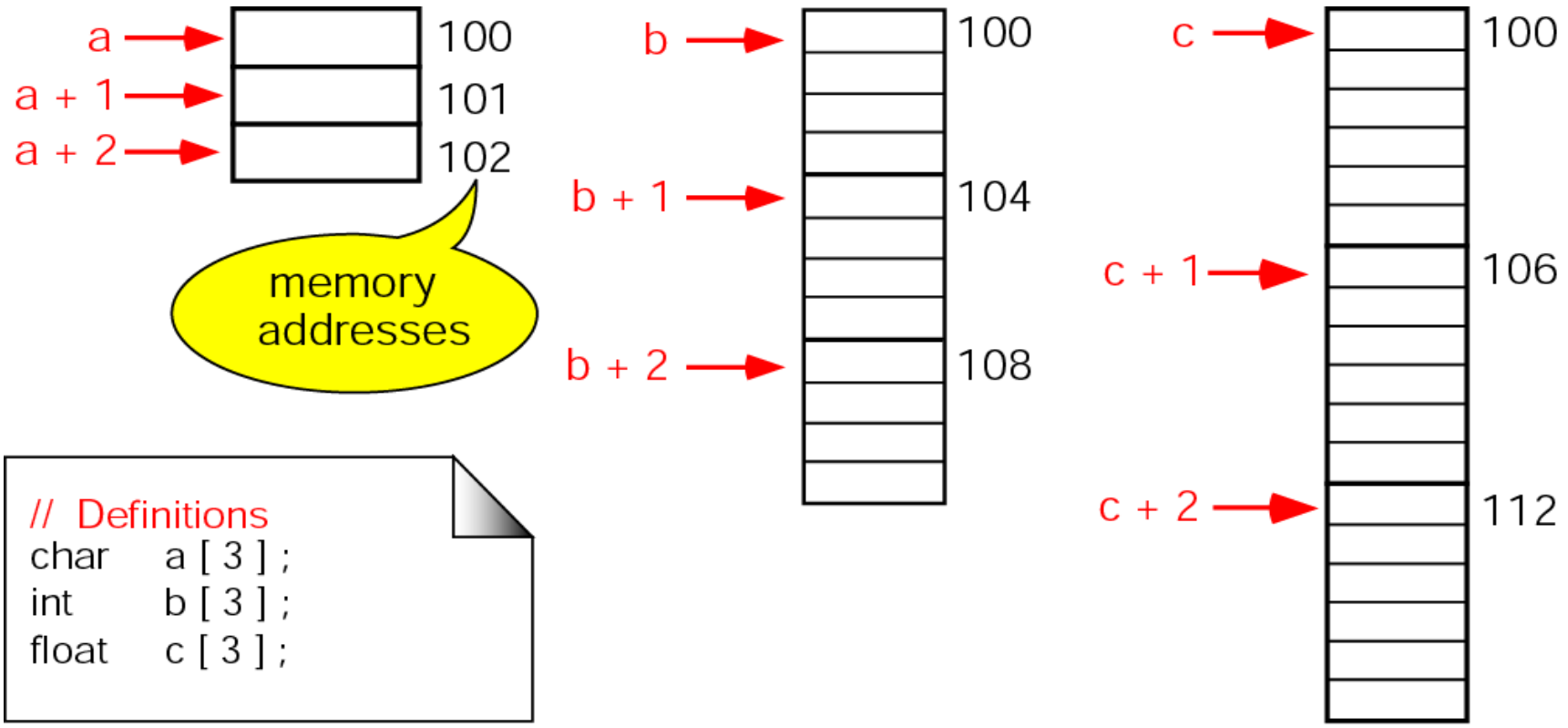


Figure 9-31 Dereferencing array pointers

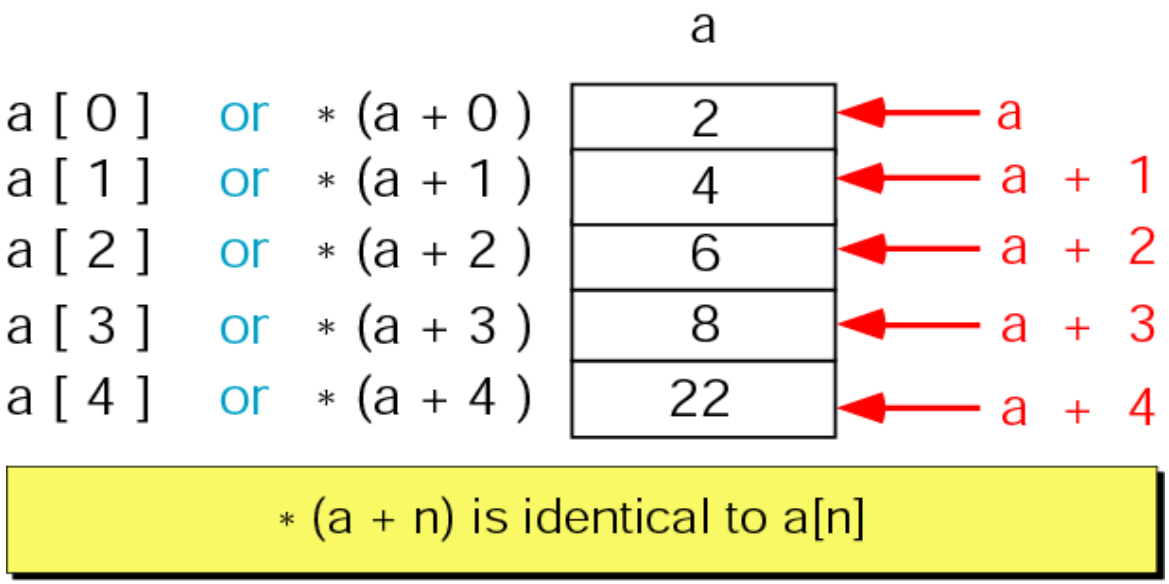
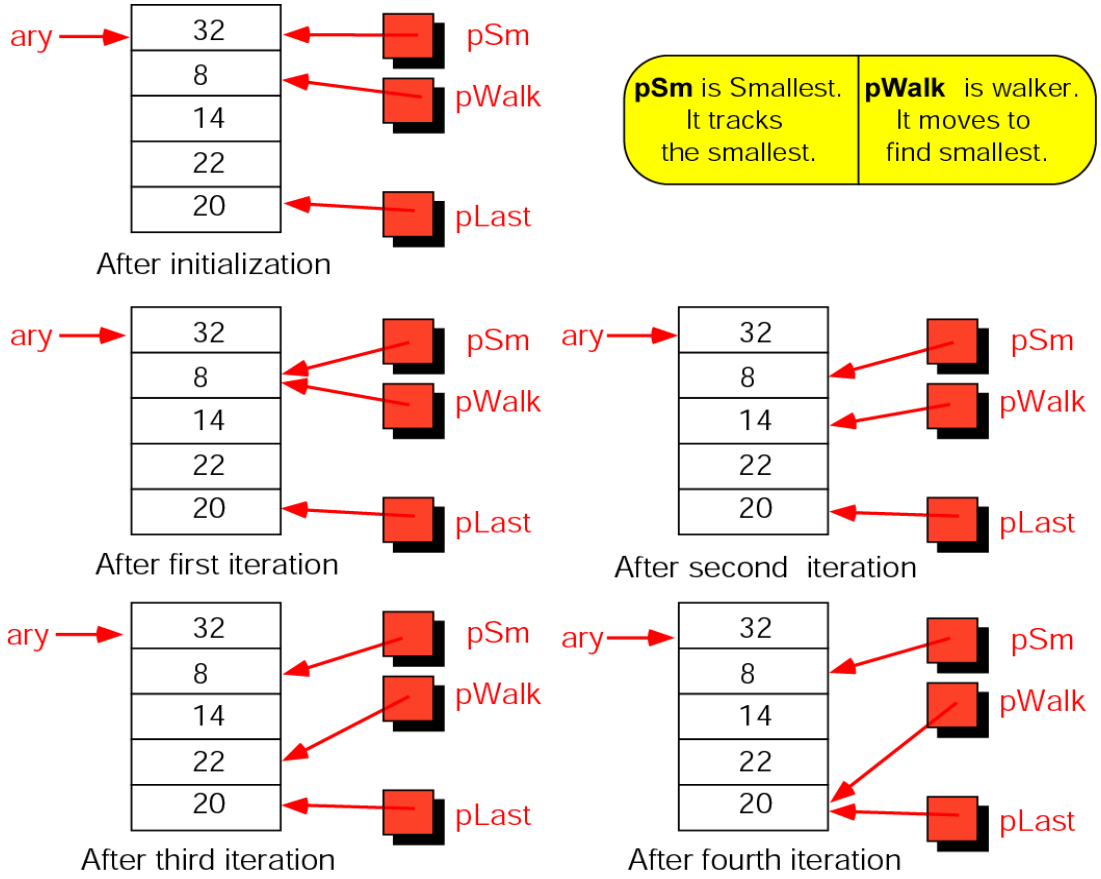
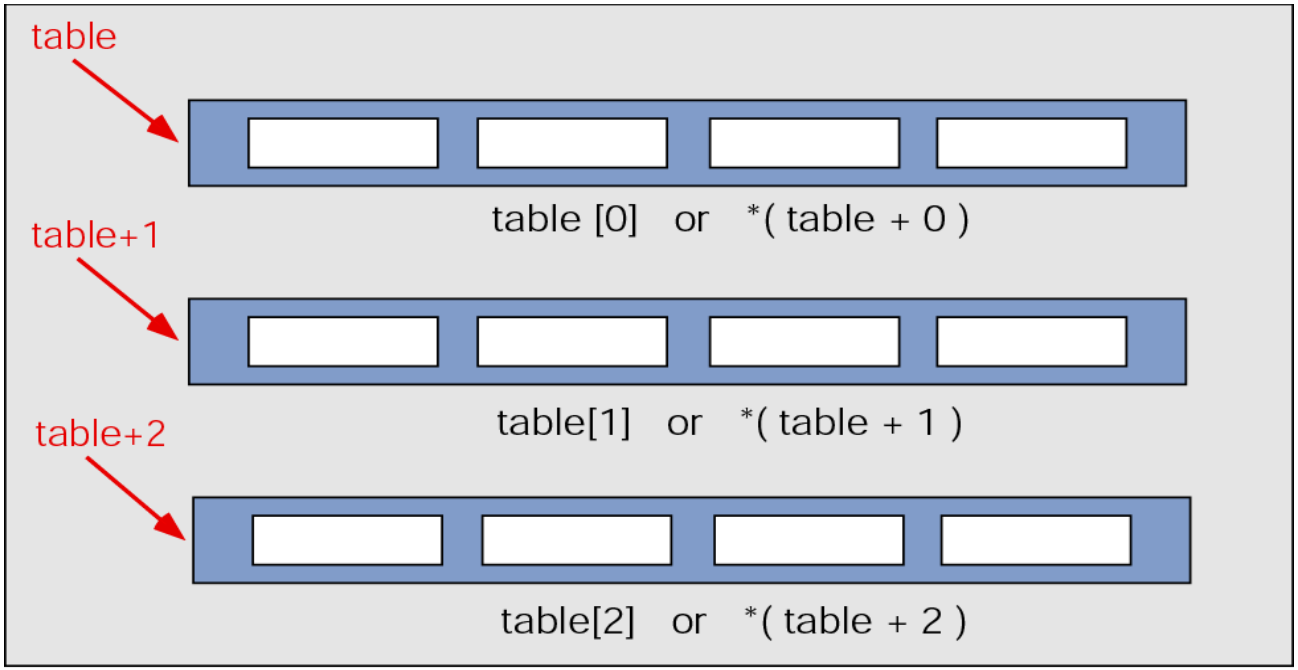


Figure 9-32 Find smallest




```
pLast = ary + arySize – 1 ;
for (int  pSm = ary, pWalk = ary + 1 ;
          pWalk <= pLast ;
          pWalk++ )
     if ( *pWalk < *pSm )
               pSm = pWalk ;
```

Figure 9-33 Pointers to two-dimensional arrays



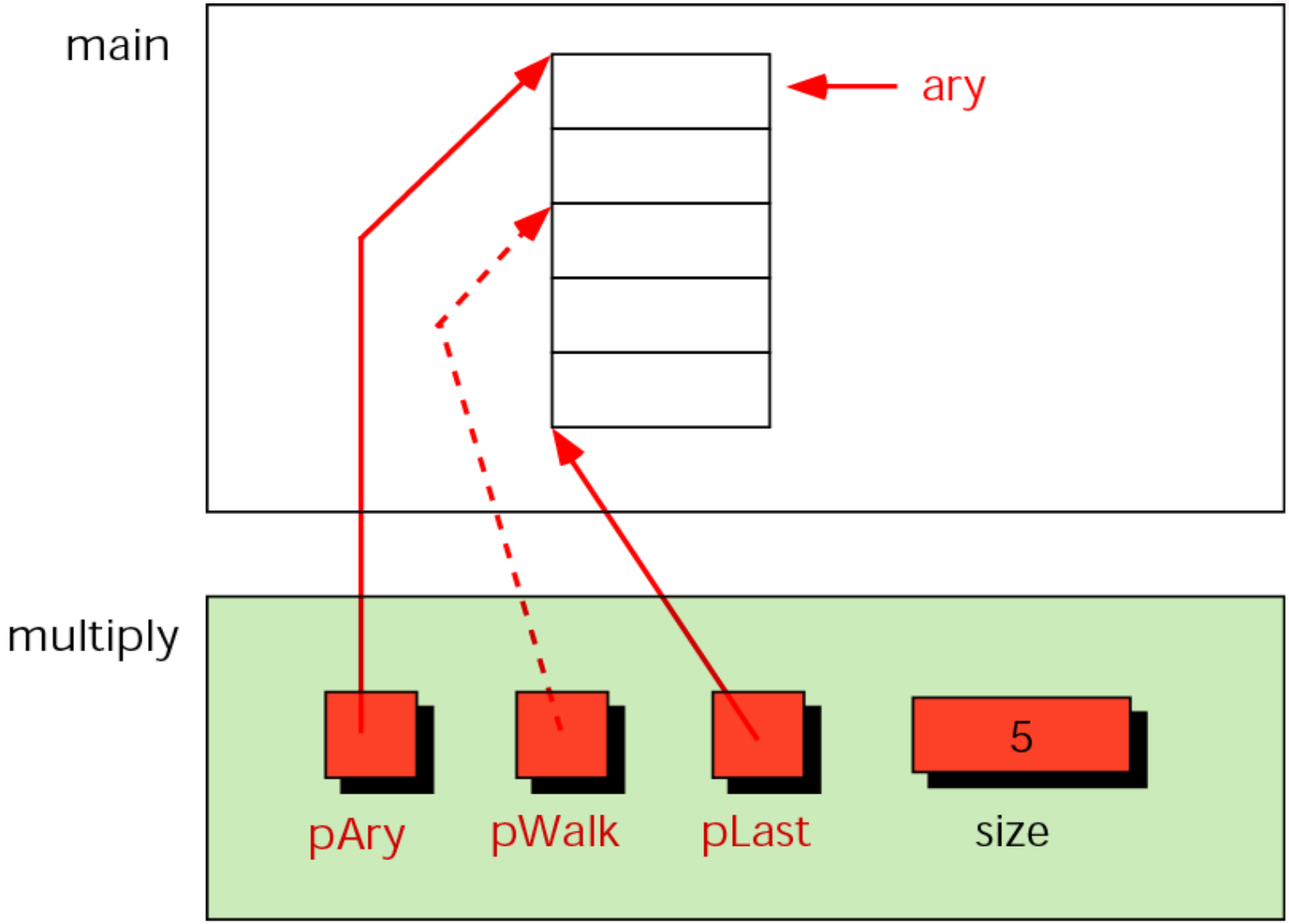
int table[3][4] ;

```
for ( int i = 0; i < 3; i++ )
    {
        for ( int j = 0; j < 4; j++ )
              int cout  <<  setw(6)
                        <<  *(*(table + i ) + j ) ;
        cout << endl ;
    }  //  for i
```

Print table

Passing an Array to a Function

Figure 9-34 Variables for multiplying array elements



MEMORY ALLOCATION FUNCTIONS

Figure 9-35 Memory allocation

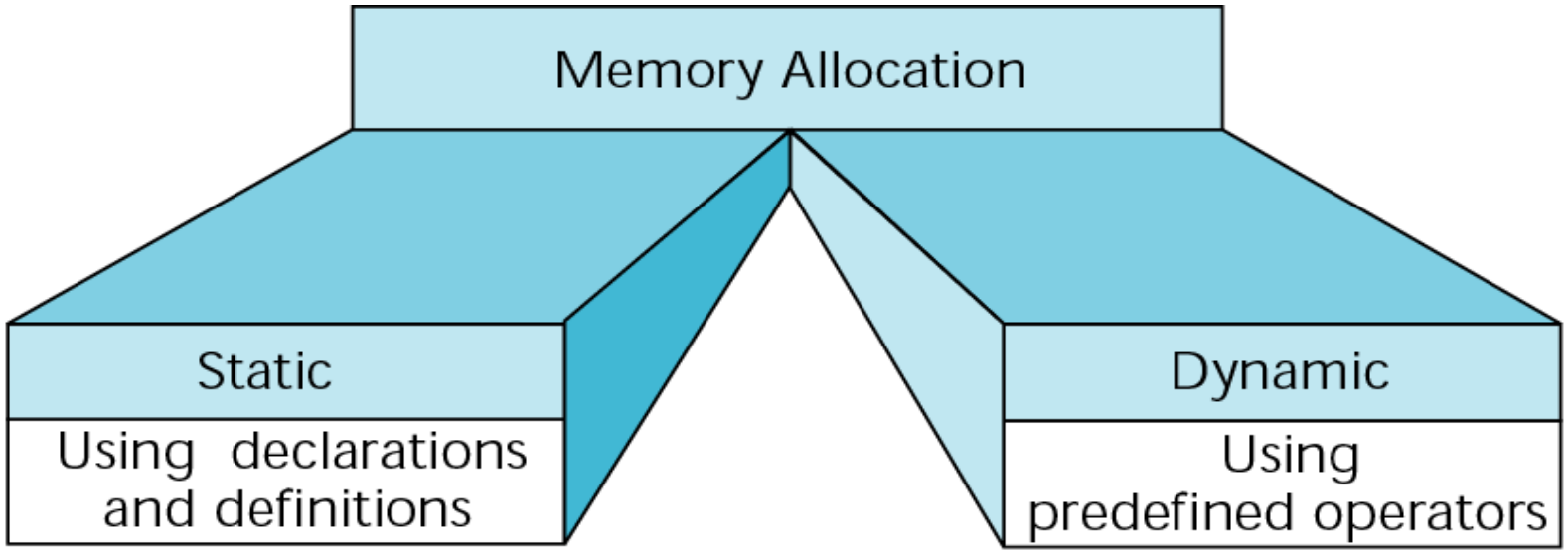


Figure 9-36 Memory management functions

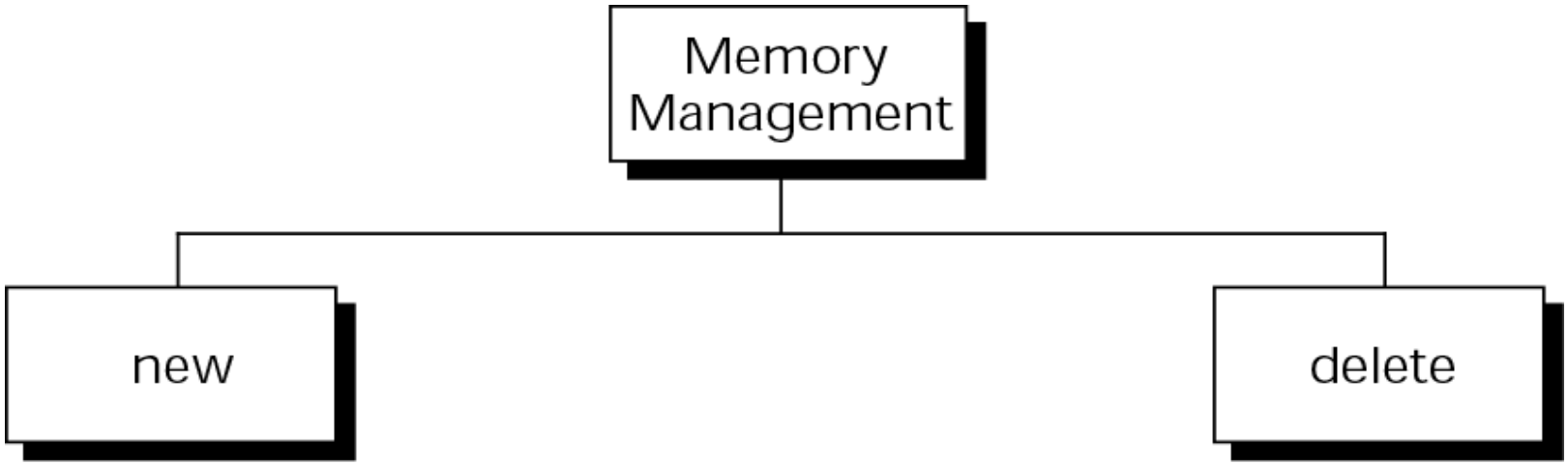
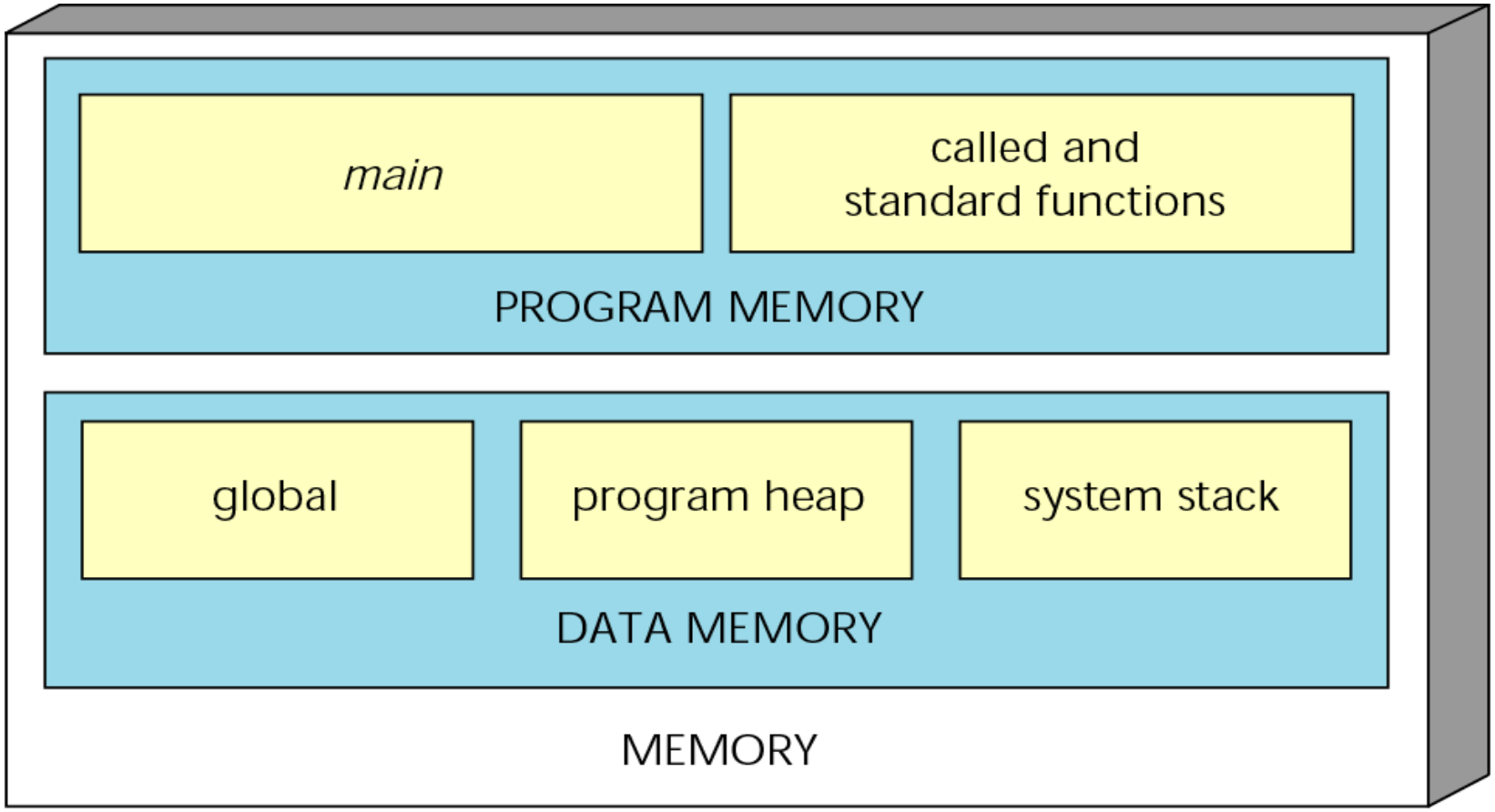


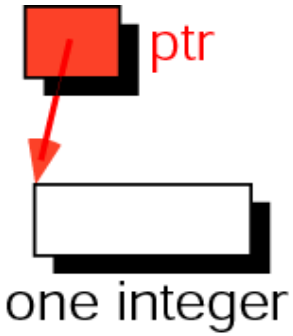
Figure 9-37 A conceptual view of memory



Note:

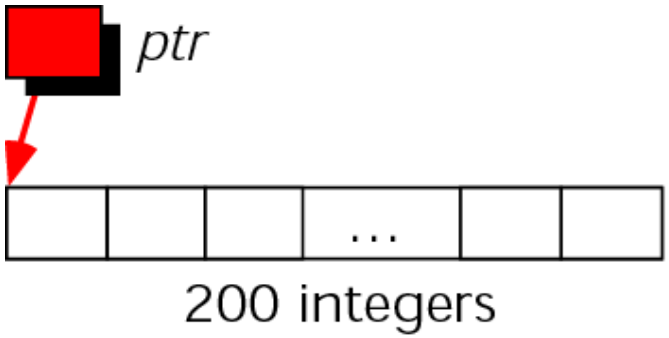
You can refer to dynamic memory only through a pointer.

Figure 9-38 *new* memory allocation for a single data item



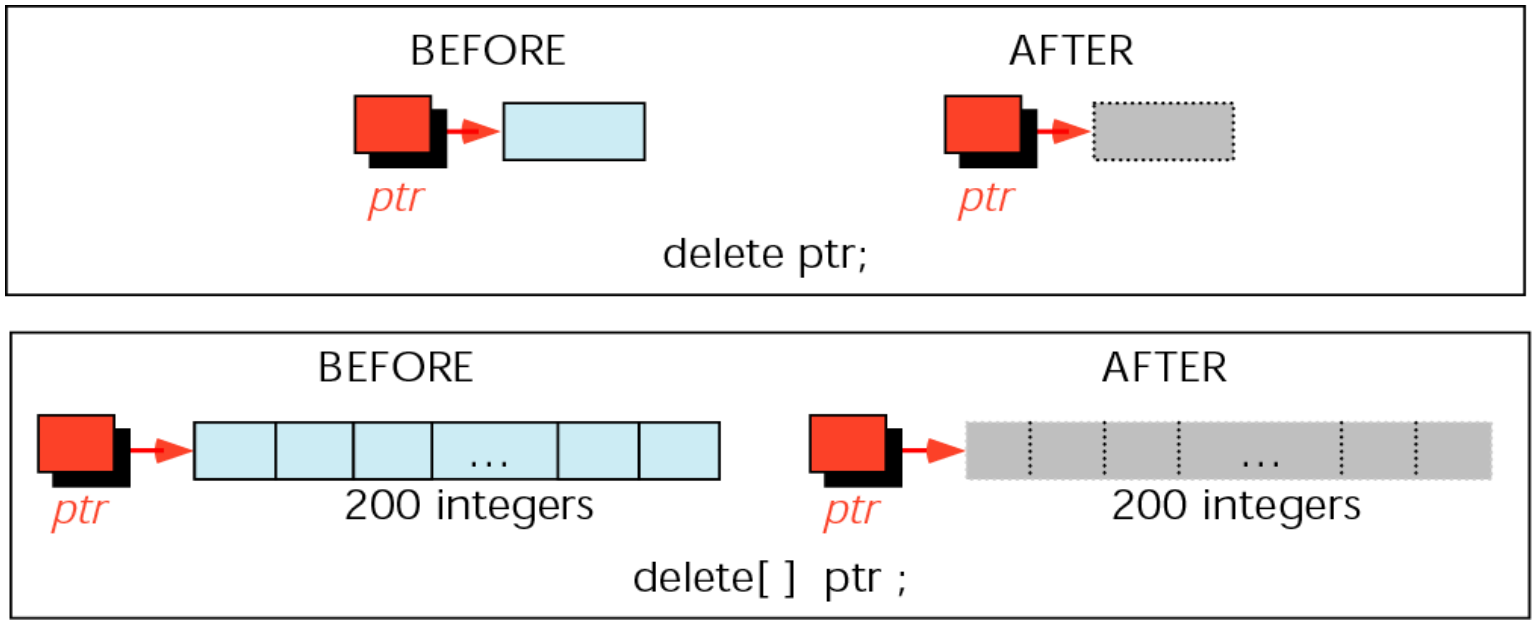

```
int* ptr = new int;
```

Figure 9-39 Memory allocation for an array




```
int* ptr = new int[200];
```

Figure 9-40 Freeing memory

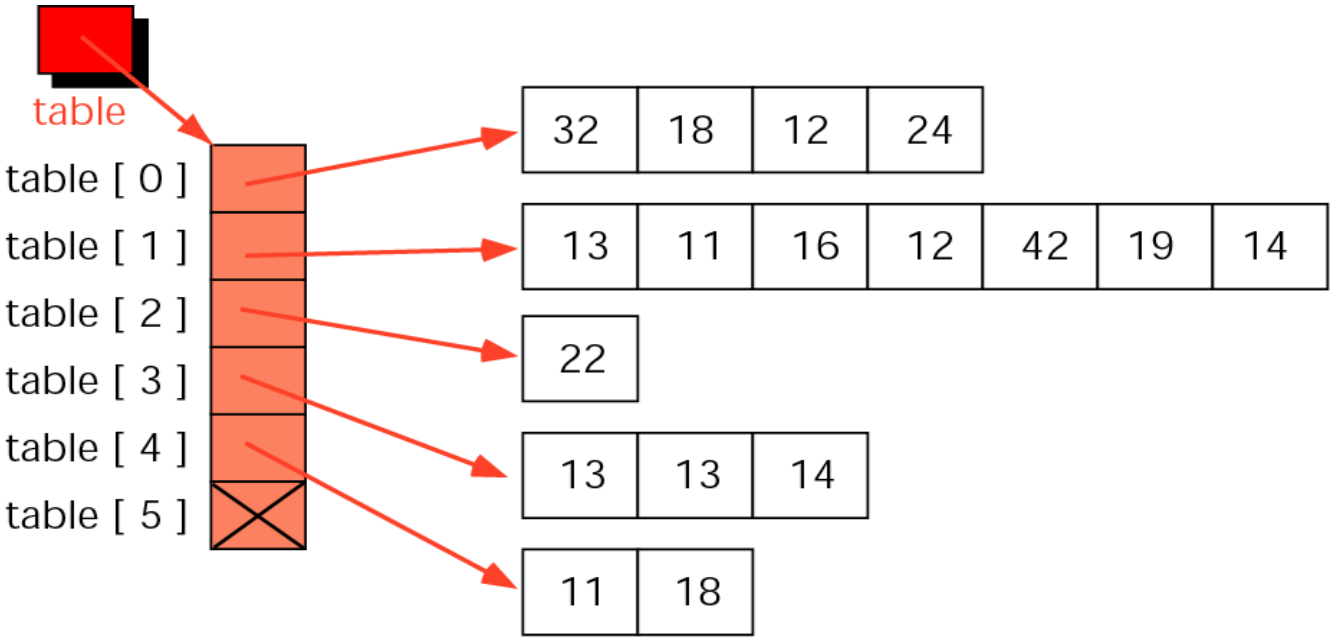


Note:

Memory allocated by new must be released with delete, and memory allocated by new[…] must be released with delete[ ].

Array of Pointers

Figure 9-41 A ragged array




```
int** table;
 ...
table = new int* [rowNum + 1] ;
 ...
table[0] = new  int[4];
table[1] = new  int[7];
table[2] = new  int[1];
table[3] = new  int[3];
table[4] = new  int[2];
table[5] = NULL ;
```

PROGRAMMING APPLICATION

Figure 9-42 Selection sort with pointers

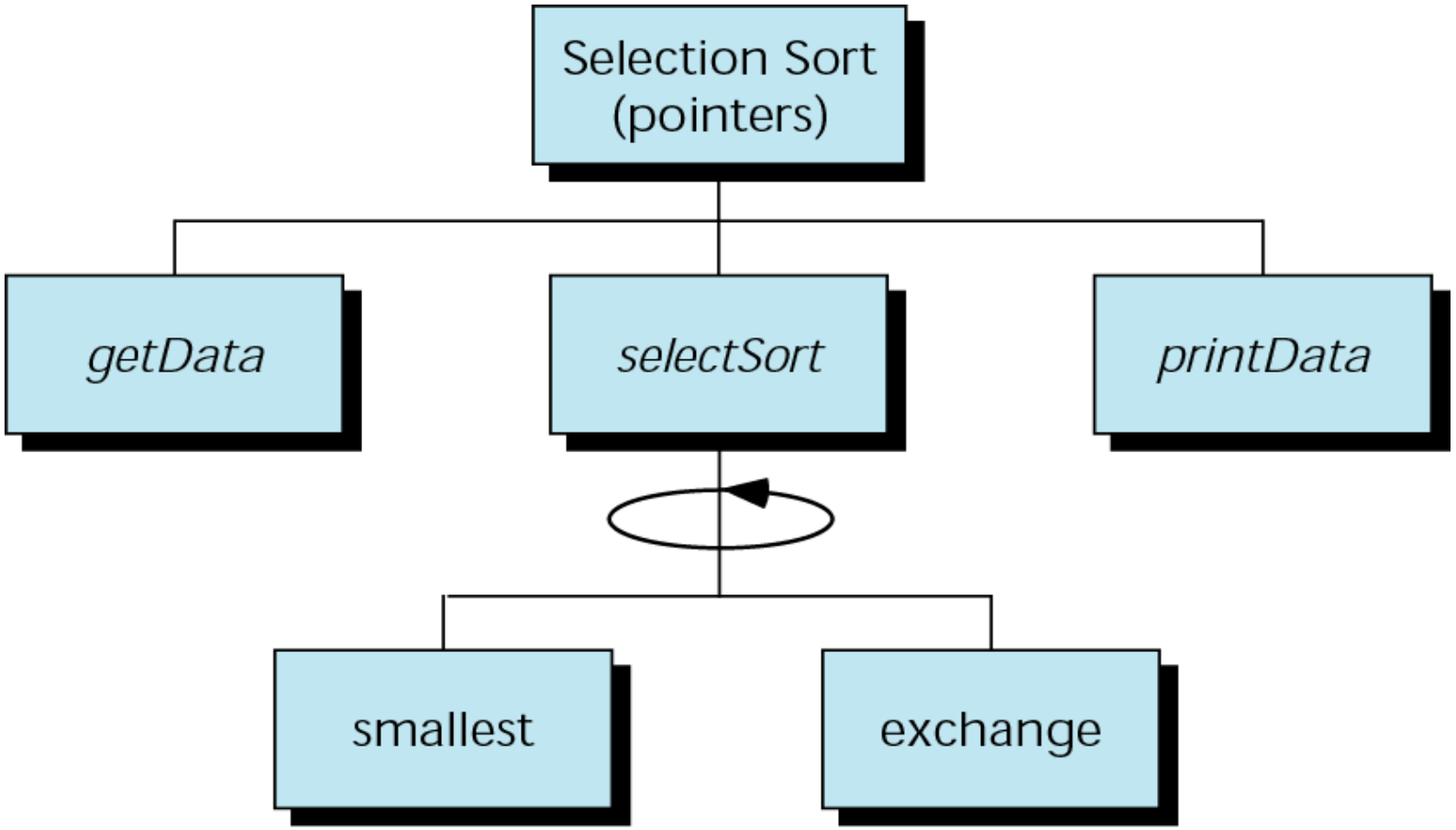


Figure 9-43 Dynamic array structure chart

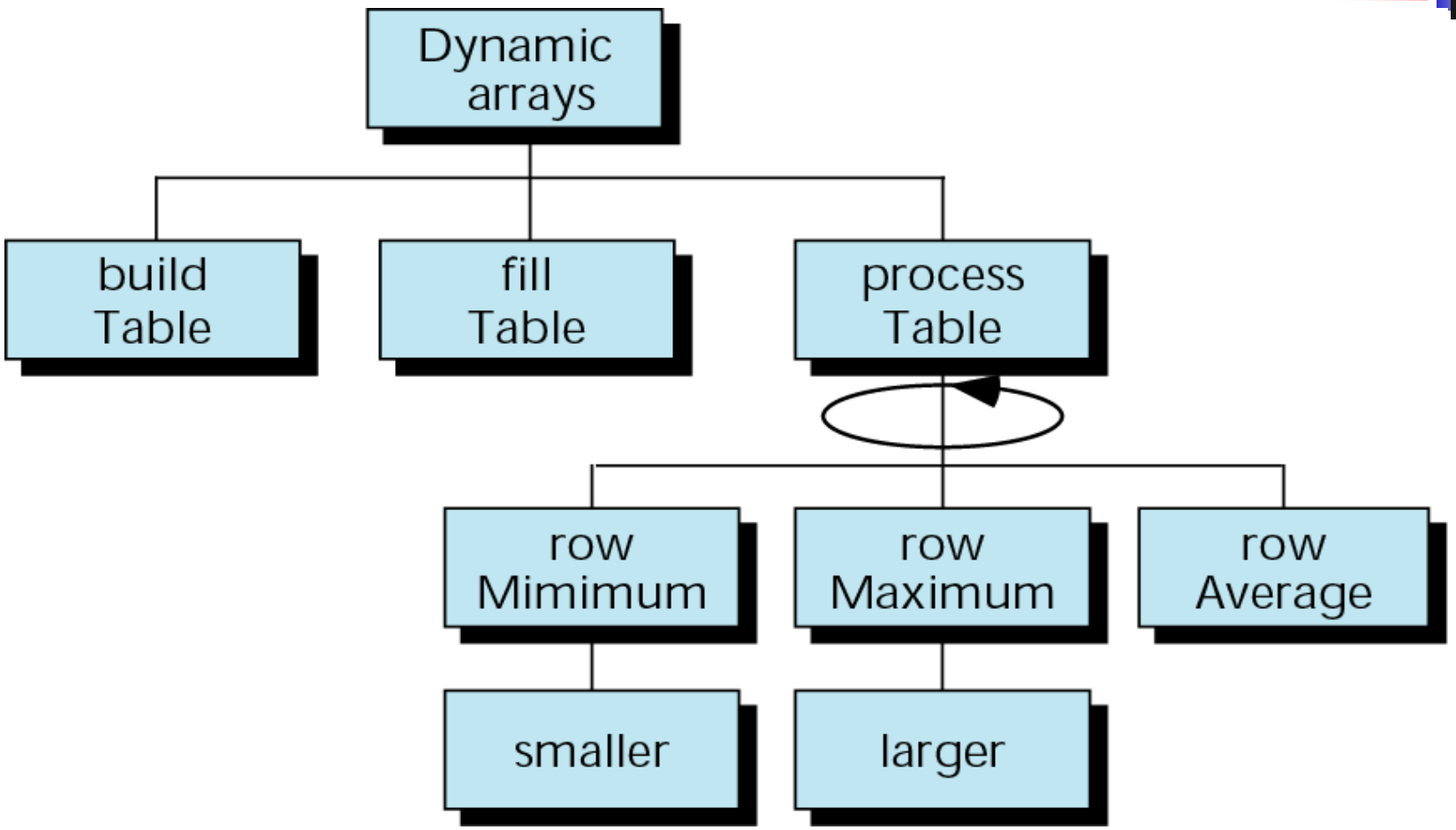
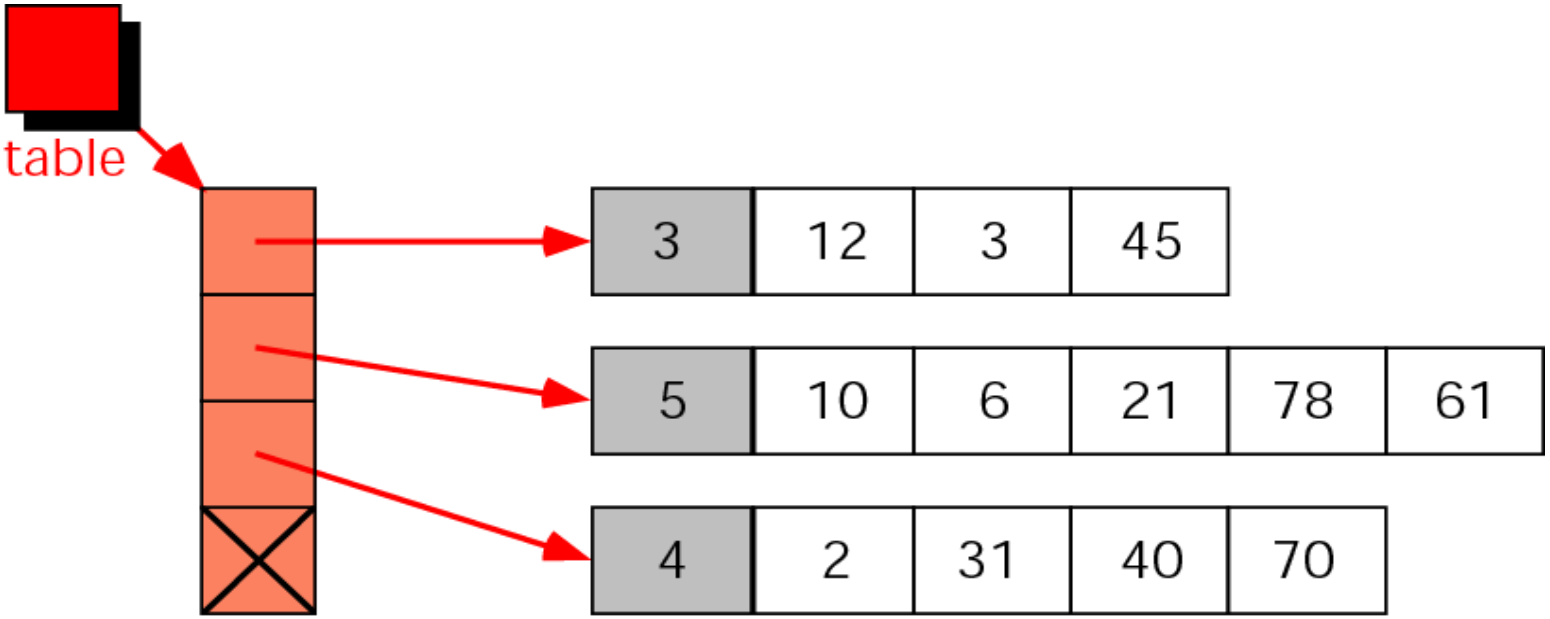


Figure 9-44 Ragged array structure



Software Engineering and Programming Style

Note:

Use value parameters when possible.