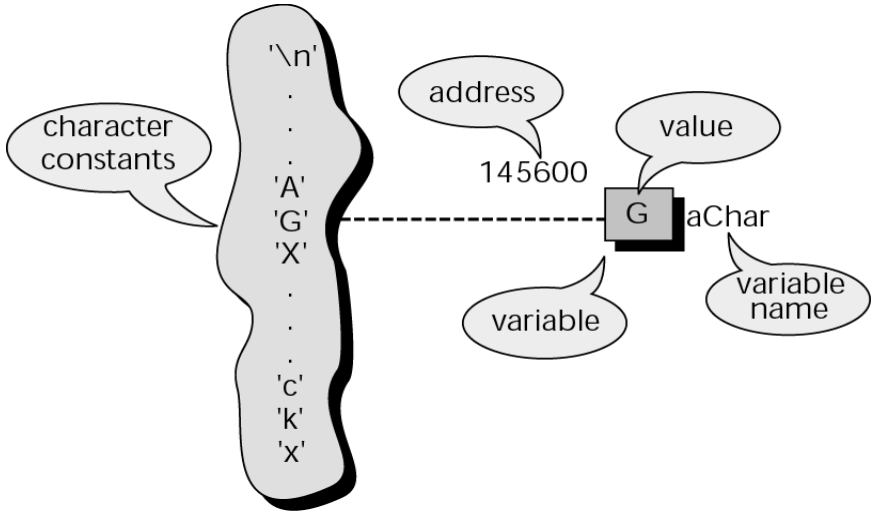
Introduction to Visual Basic and Visual C++

Lesson 11

Pointer

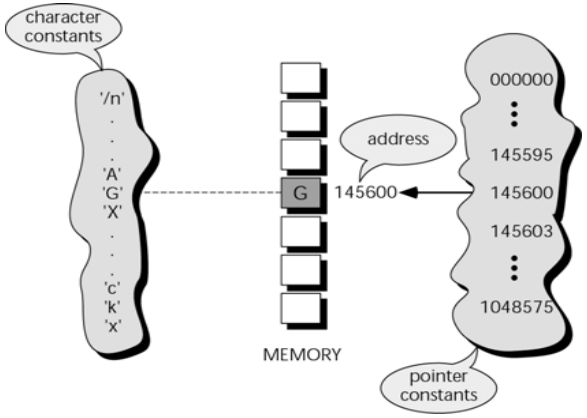
Memory Allocation and Management

Character Constants and Variables



Pointer Constants

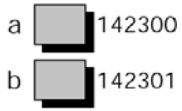
- Pointer constants, drawn from the set of addresses for a computer, exist by themselves. We cannot change them; we can only use them.



Print Character Addresses

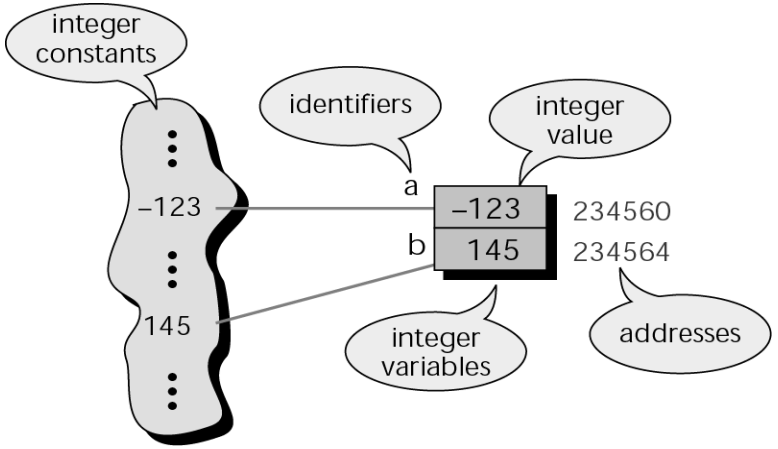
- When the ampersand (&) is used as a prefix to a variable name, it means "address" of variable. When it is used as a suffix to a type, it means reference parameter.

```
// This program prints character addresses
#include <iostream>
using namespace std;
int  main  ( )
{
  char   a ;
  char   b ;
  cout << &a << &b ;
  return 0 ;
} //  main
```

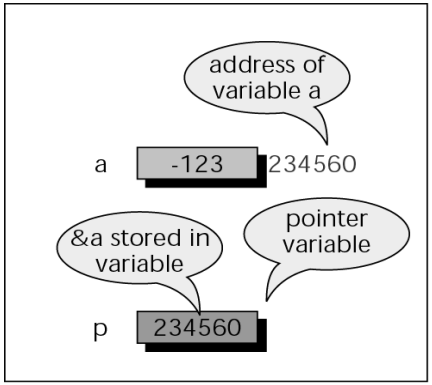


Integer Constants and Variables

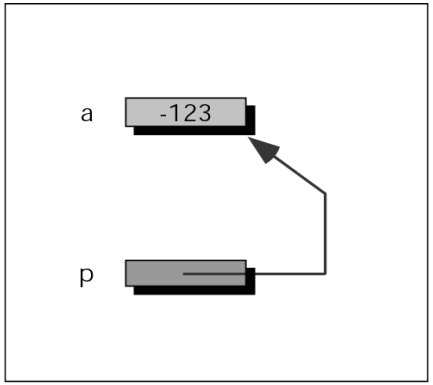
- The address of a variable is the address of the first byte occupied by that variable



Pointer Variable

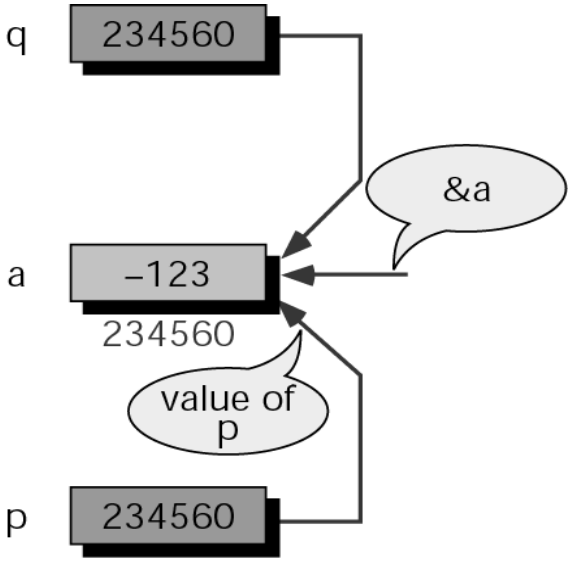


Physical representation

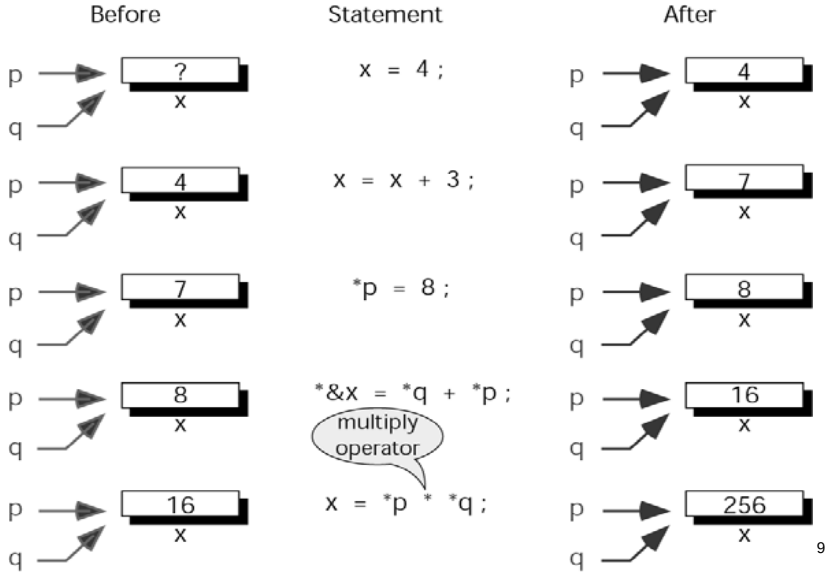


Logical representation

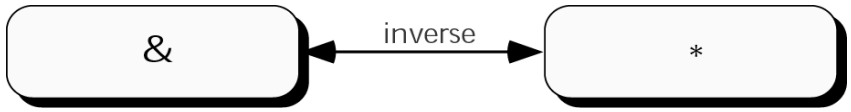
Multiple Pointers to a Variable



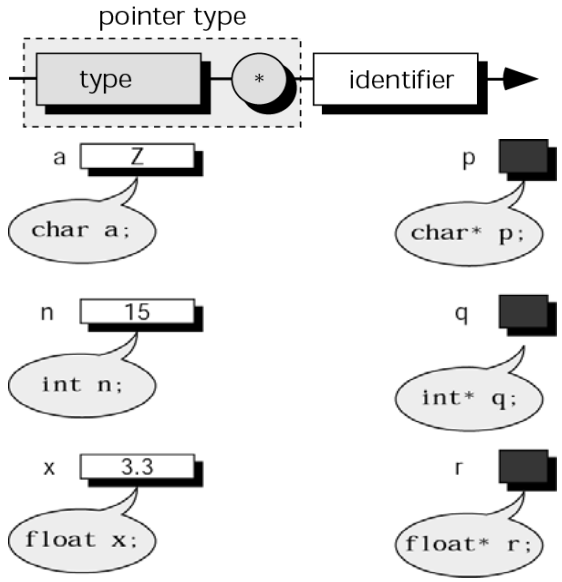
Accessing Variables through Pointers



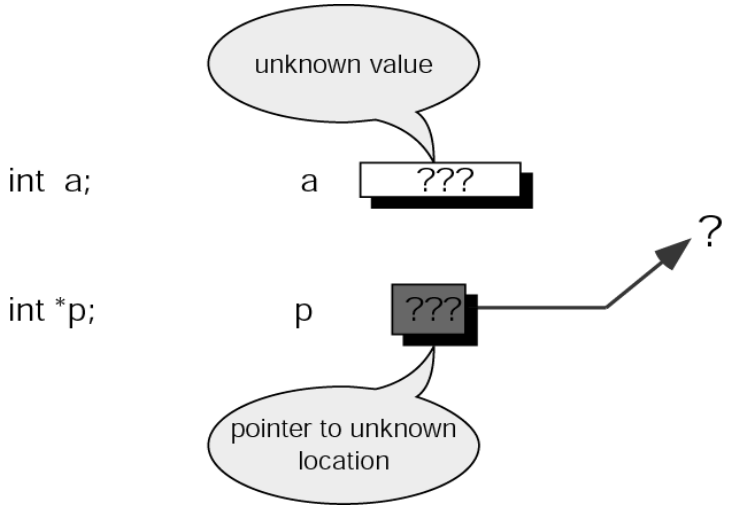
Address and Indirection Operators



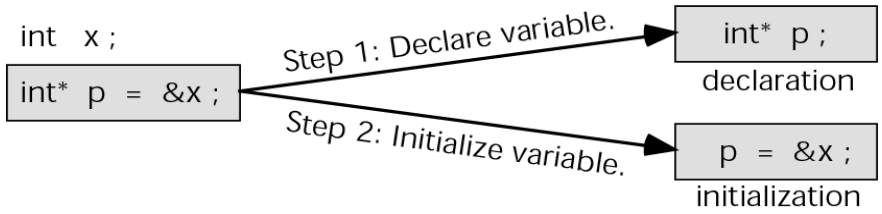
Pointer Variable Declaration



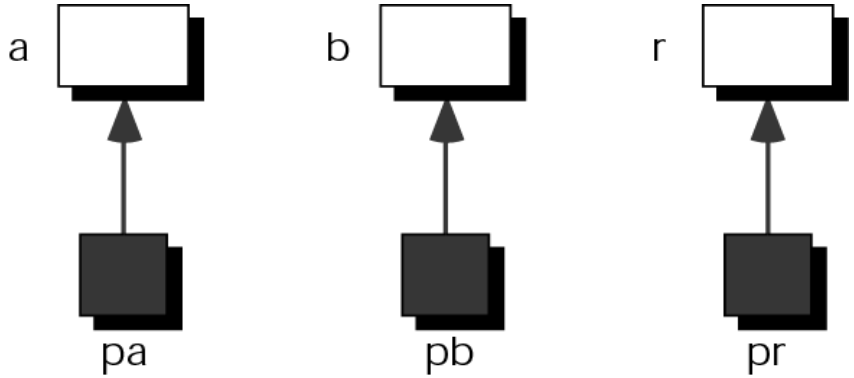
Uninitialized Pointers



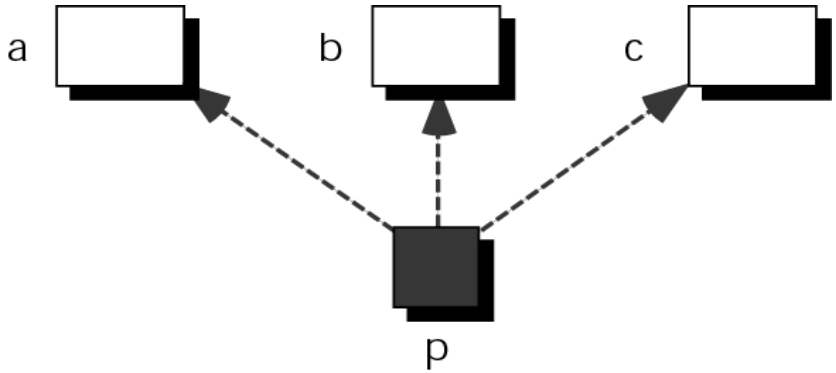
Initializing Pointer Variables



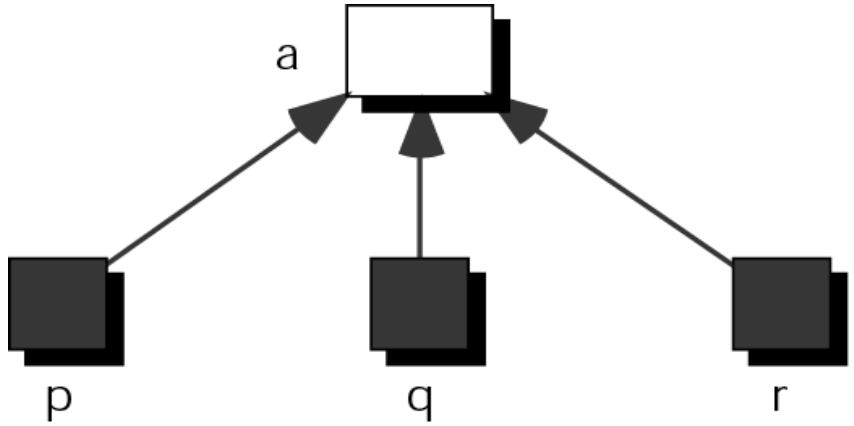
Add Two Numbers Using Pointers



Demonstrate Pointer Flexibility



Using One Variable with Many Pointers



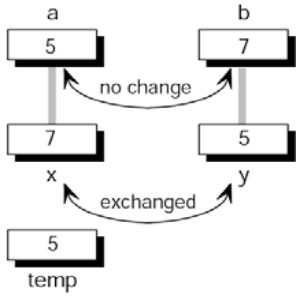
Example: Exchanging Values

```
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



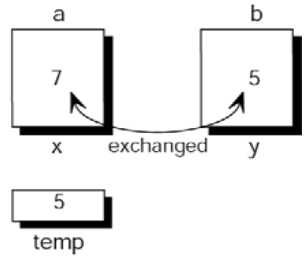
Example: Exchanging Values

```
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



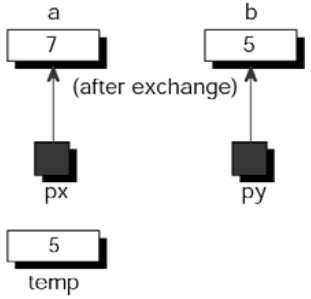
Example: Exchanging Values

```
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

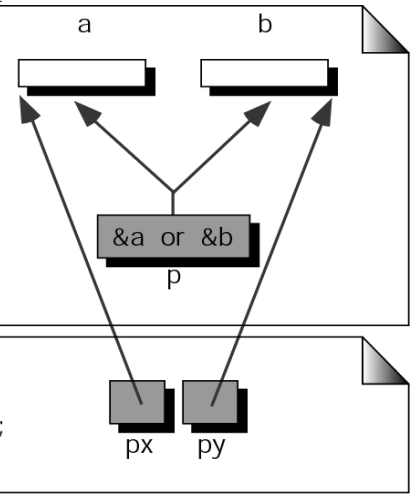


Functions Returning Pointers

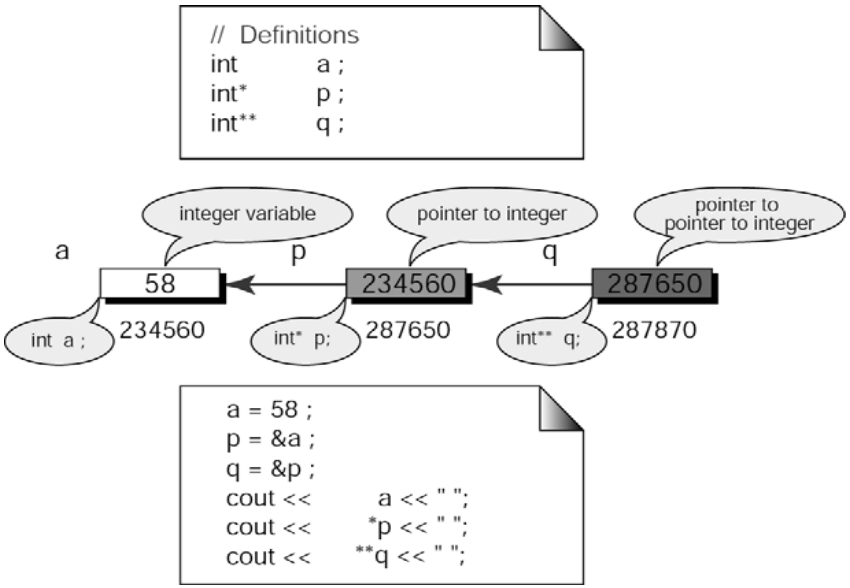
- It is a serious error to return a pointer to a local variable.

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

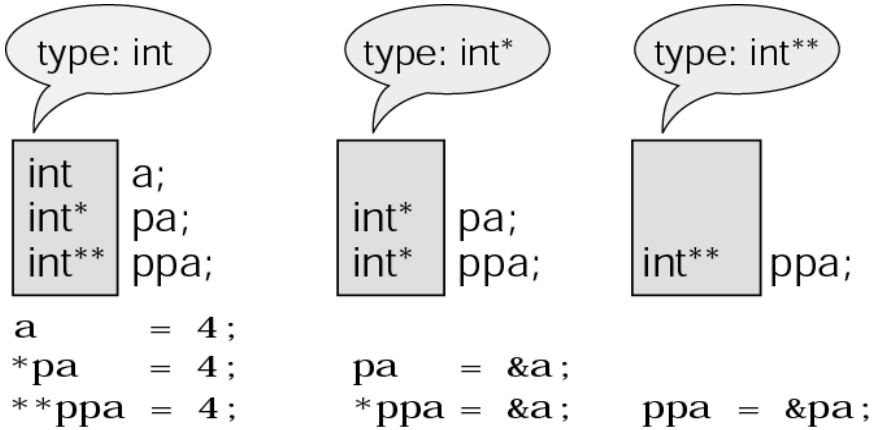
```
int*  smaller (int* px, int* py )
{
  return ( *px < *py ? px : py ) ;
} // smaller
```



Pointers to Pointers

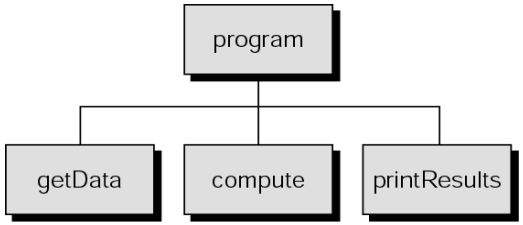


Pointer Types must Match

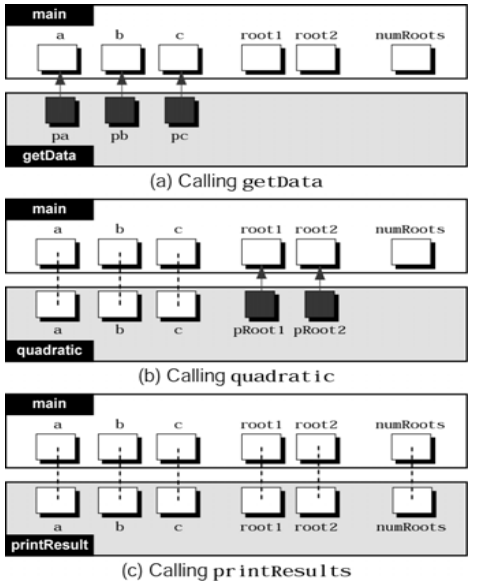


Common Program Design

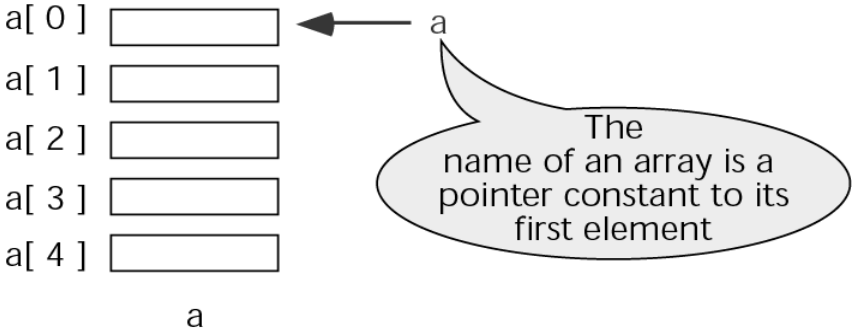
- Create local variables when a value parameter will be changed within a function so that the original value will always be available for processing.
- When several values need to be sent back to the calling function, use address parameters for all of them. Do not return one value and use address parameters for the others.



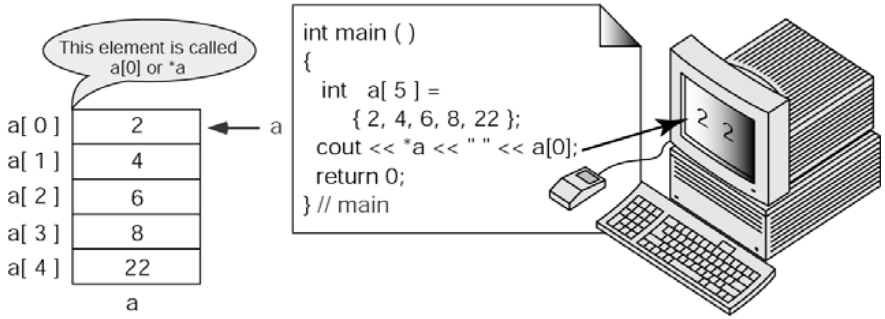
Using Pointers as Parameters



Pointers to Arrays

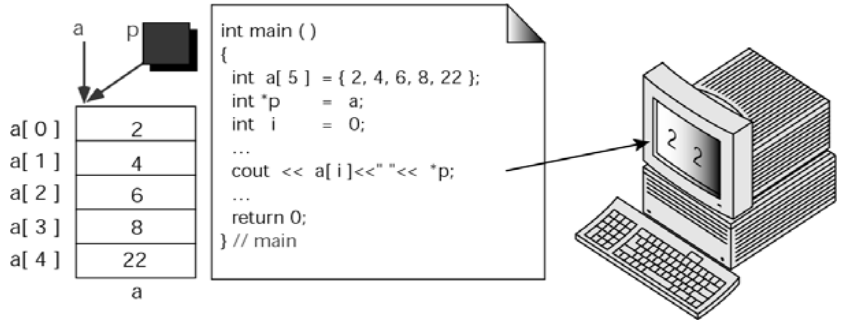


Dereference of Array Name

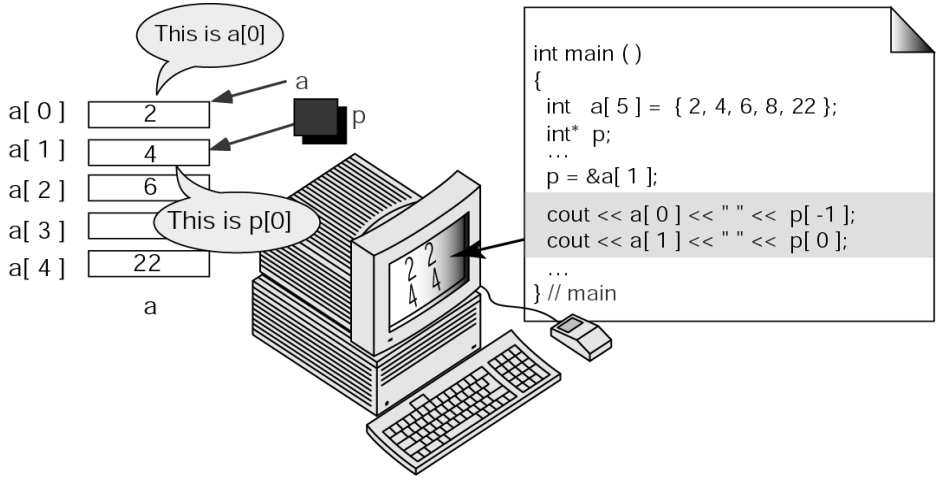


Array Names as Pointers

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

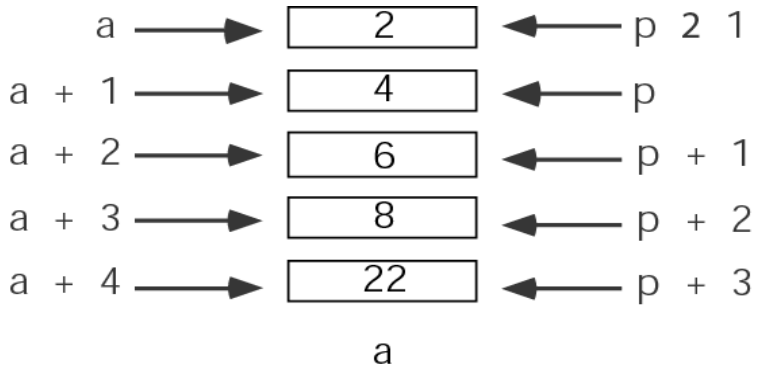


Multiple Array Pointers

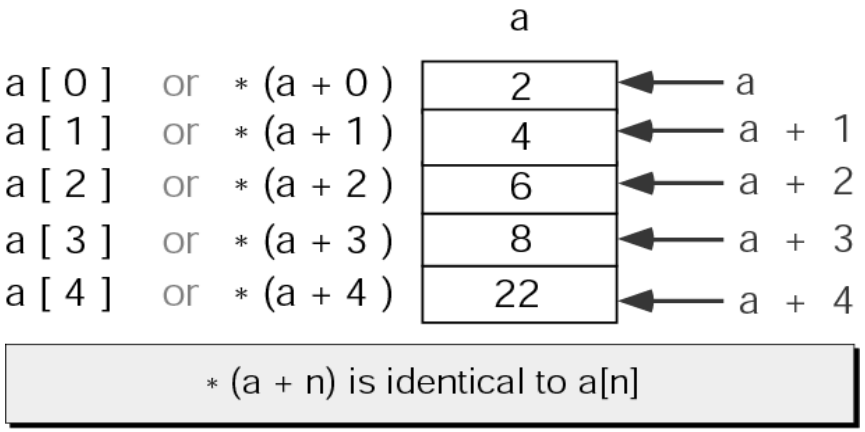


Pointer Arithmetic

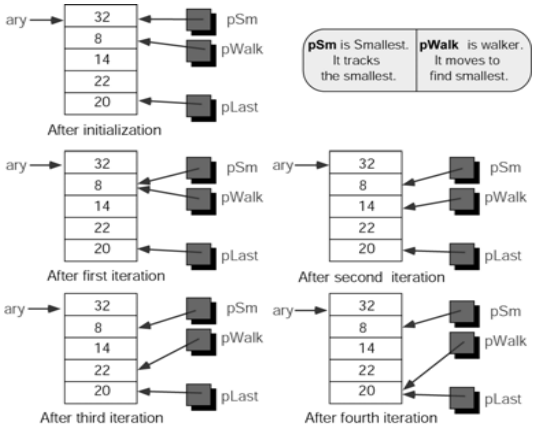
- Given pointer, p, p ± n is a pointer to the value n elements away



Dereferencing Array Pointers

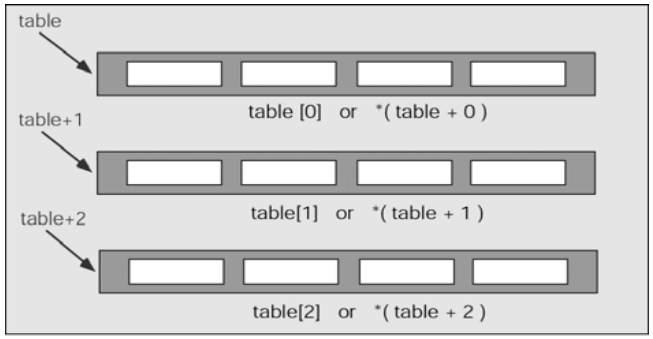


Example: Find Smallest




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


Pointers to Two-Dimensional Arrays

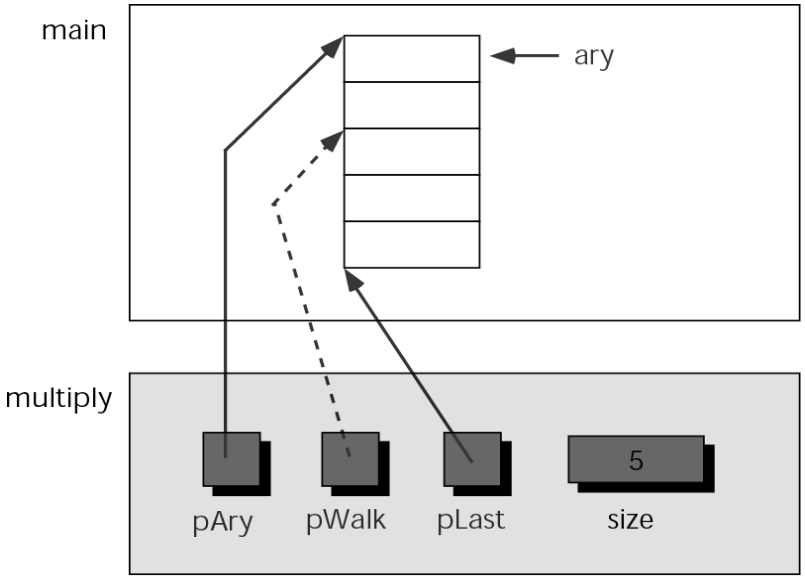



```
int  table[3][4] ;
```

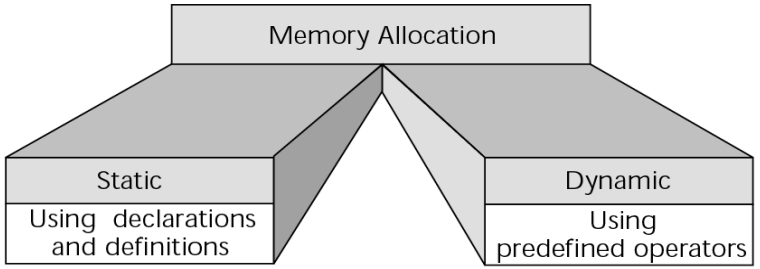
```
for ( int i = 0; i < 3; i++ )
    {
        for ( int j = 0; j < 4; j++ )
            int cout  <<  setw(6)
                      <<  *(*(table + i ) + j ) ;
        cout << endl ;
    }  // for i
```

Print table

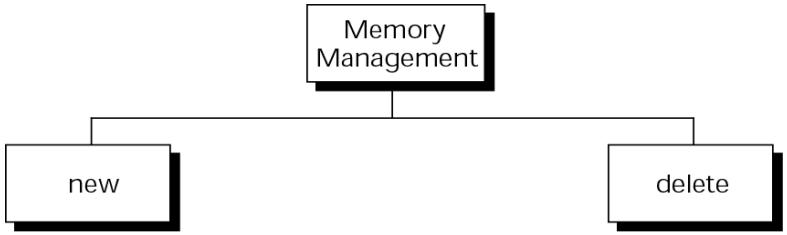
Variables for Multiplying Array Elements



Memory Allocation

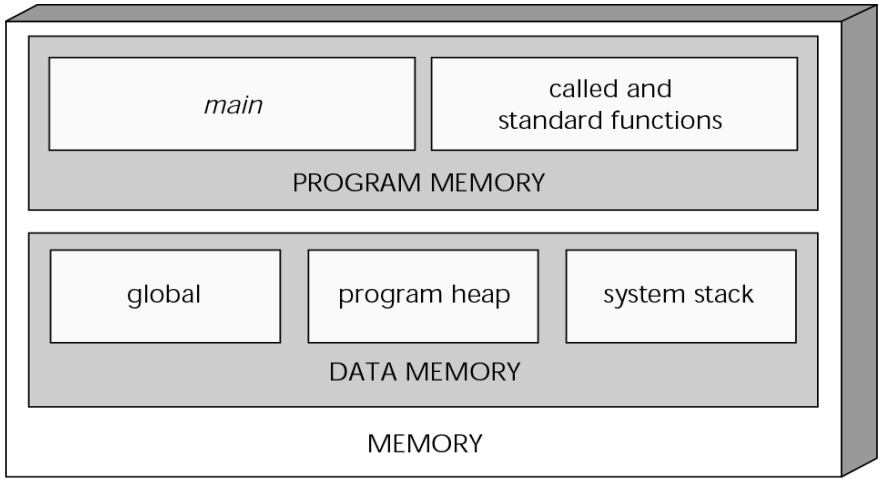


Memory Management Functions

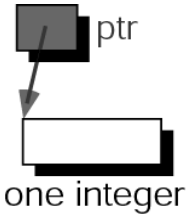


A Conceptual View of Memory

- You can refer to dynamic memory only through a pointer

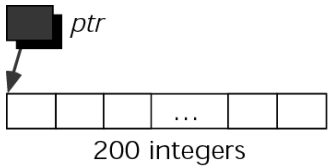


New Memory Allocation for a Single Data Item




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

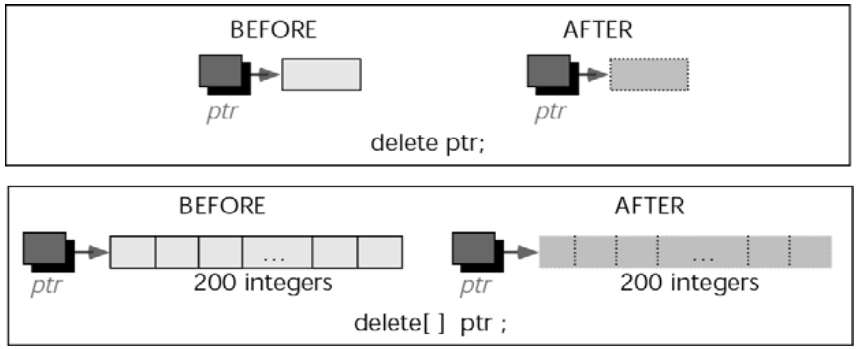

Memory Allocation for an Array



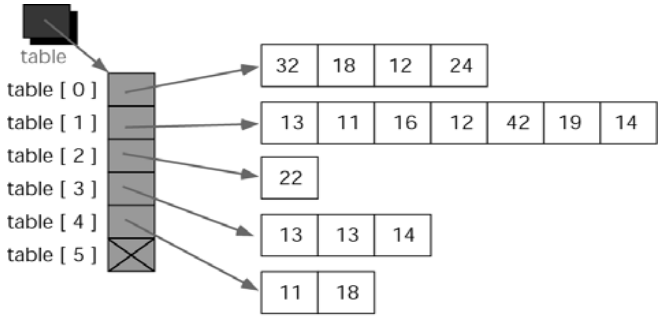

```
int* ptr = new int[200];
```


Freeing Memory

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



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 ;
```