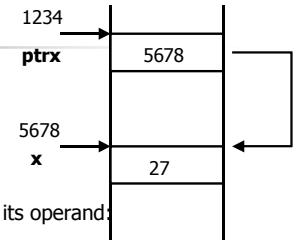
Lecture 3 - Data Structures

Pointers

Pointers



address operator &: returns the address of its operand:

scanf*("%d, %d", &first, &second) ;*

Declaration and initialization

int *x = 27 ;*

- allocate memory for **int**, referenced by **x**.
- initialize with 27

pointer – variable which contains the address of some other variable:

```
int *ptrx ;
ptrx = &x ;
```

```
int *ptrx = &x;
```

- **ptrx** is a variable with its own address (e.g. 1234).**ptrx** point to the variable **x**.

* - Unary **indirection operator** (**dereference** operator)

- *has one operand* – address *(pointer)*
- *access* the contents of its operand (the contents of memory location, the *pointer points at).*

```
int y = 3, x = 2 ;
int *ptry = &y, *ptrx = &x ;
        y = x ;
        y = *ptrx ;
        *ptry = *ptrx ;
        *ptry = x ;
```

If **ptrx** is a pointer to integer, ***ptrx** can be used in any context where an integer could be found:

*y = *ptrx + 1 ;* ***printf****("The value of x is %d\n", *ptrx) ;*

Indirection with the increment and decrement:

*(*ptrx)++ ;*

The parentheses are necessary - *both operators* are of *equal precedence* and *associate right to left*.

Without the parentheses:

**ptrx++ ;*

would *increment* **ptrx** *then* access the *value*.

Pointers are variables and can be manipulated like other variables.

```
int x ;
int *ptrx, *ptry ;
ptrx = &x ;
```

then:

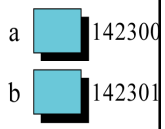
```
ptry = ptrx ;
```

copies the content of **ptrx** into **ptry** (**ptrx** and **ptry** point to x).

double *x ; ptrx =* ***&x*** *;* // illegal - pointer to integer

Accessing Variables through Pointers

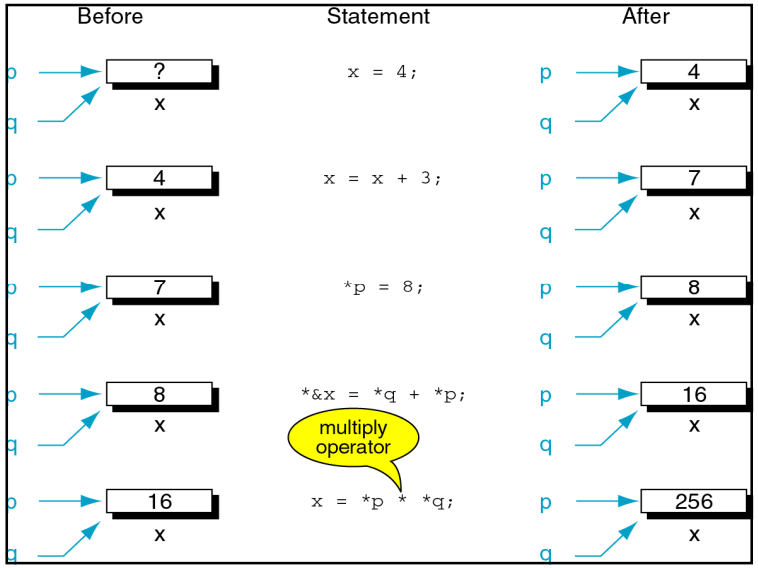
```
// This program prints character addresses
#include <iostream.h>

int  main  ( void )
{
// Local Declarations
   char   a ;
   char   b ;

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



5

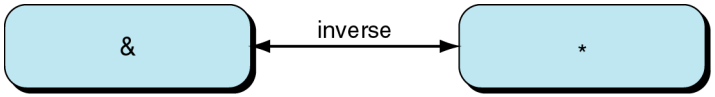
```
int a=6 , *p=&a ;

a++ ; a = a + 1 ; *p = *p + 1 ;

(*p)++ ; *p++ ; *&a = 4 ;
```

6

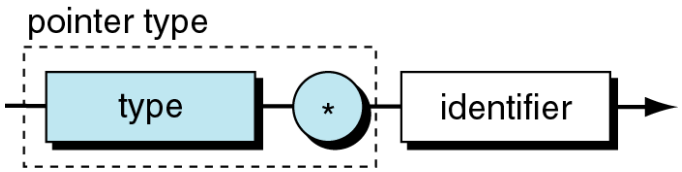


Address and Indirection Operator

***(&x)**

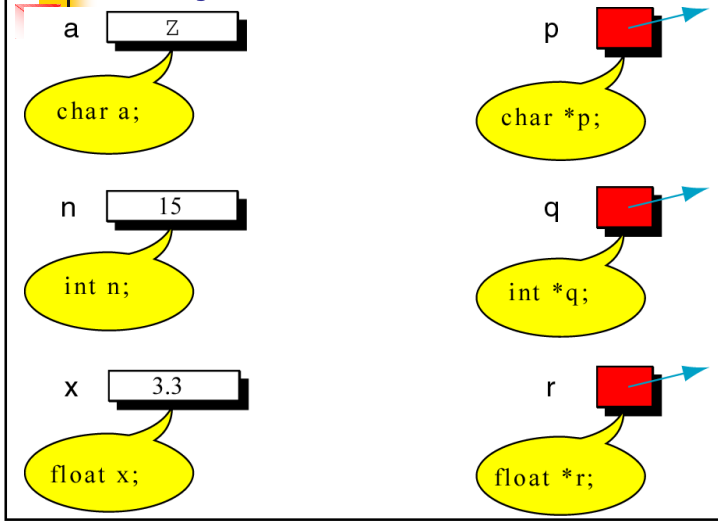


- Pointer Variable Declaration



8

Declaring Pointer Variables



9

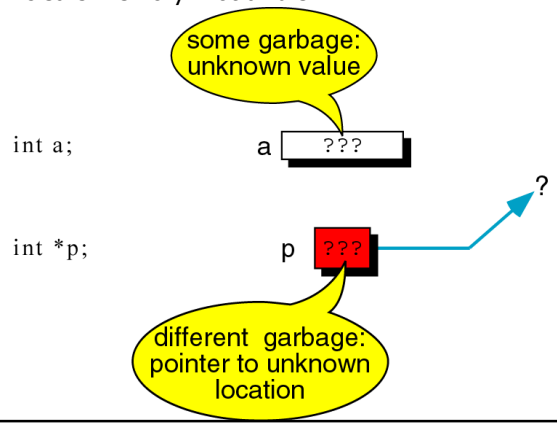
// P09-01 Demonstrate use of pointers

```
#include <iostream.h>
int main (void)
{
// Local declarations
   int  a;
   int *p;
// Statements
   a = 14;
   p = &a;
   cout << a << " " << &a << endl;
   cout << p << " " << *p << " " << a << endl;
   return 0;
}  // main
/*
Results:
   14            0x00d50a2a
   0x00d50a2a    14    14
*/
```

10

Initialization of Pointer Variables

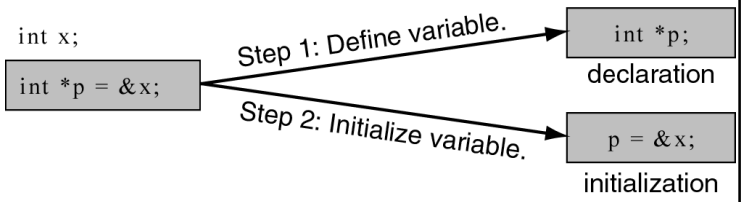
- *Uninitialized pointers-* generate *run-time error*. C++ does not initialize variables (in general). OS often clears memory- not a rule.



11

Initializing Pointer Variables

int *p = NULL ;

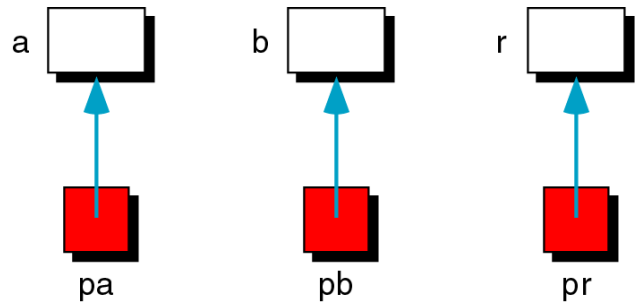


12

```
// P-09-02 Fun with pointers
#include <iostream.h>
int main (void)
{ //      Local Declarations
    int  a, b, c, *p, *q, *r ;
// Statements
    a = 6;              b = 2;          p = &b;
    q = p;              r = &c;         p  = &a;
    *q = 8;             c  = *q;        *r = *p;
    *r = a + *q + *&c;
    cout <<  a << " " <<  b << " " <<  c << endl;
    cout << *p << " " << *q << " " << *r << endl;
    return 0;
}  // main
/*
Results:
    6 8 20
    6 8 20
*/
```

13

Add two numbers using pointers

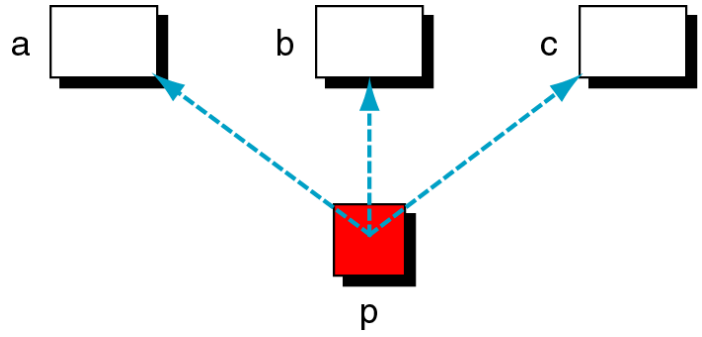


14

```
// p09_03.cpp Add two numbers using pointers
#include <iostream.h>
int main (void)
{ //      Local Declarations
    int  a;             int  b;         int  r;
    int* pa = &a;       int* pb = &b;   int* pr = &r;
// Statements
    cout << "\nEnter the first number: "; cin  >> *pa;
    cout << "\nEnter the second number: "; cin  >> *pb;
    *pr = *pa + *pb;
    cout << endl;
    cout << *pa << " + " << *pb << " is " << *pr << endl;
    return 0;
}  //  main
//  Results:
//  Enter the first number: 17
//  Enter the second number: 29
//  17 + 29 is 46
```

15

One pointer can be used to access several variables:

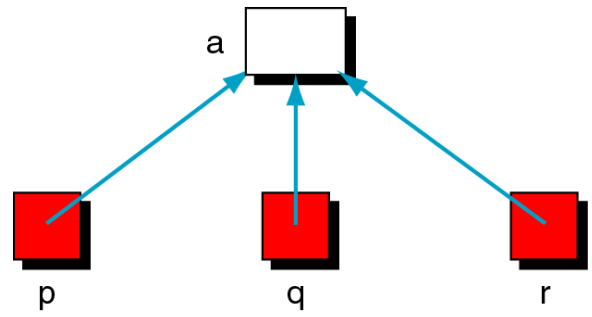


16

```
// P09-04 Using one pointer for many variables
#include <iostream.h>
#include <iomanip.h>
int main ( void )
{ //      Local Declarations
    int  a; int  b; int  c; int *p;
// Statements
    cout << "\nEnter three numbers and key return: ";
    cin  >> a >> b >> c;
    p = &a; cout << setw(3) << *p << endl;
    p = &b; cout << setw(3) << *p << endl;
    p = &c; cout << setw(3) << *p << endl;
    return 0;
}  // main
//  Results:
// Enter three numbers and key return: 5 8 -3
//  5
//  8
// -3
```

17

One Variable with many pointers



18

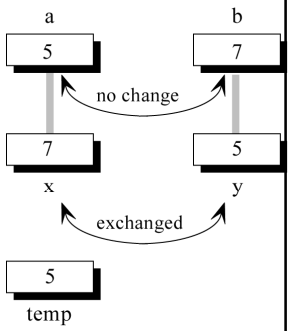
```
// P09-05 Using one variable with many pointers
#include <iostream.h>
int main (void)
{ //      Local Declarations
    int  a; int *p = &a; int *q = &a; int *r = &a;
// Statements
    cout << "\nEnter a number: "; cin  >> a;
    cout << *p << endl;
    cout << *q << endl;
    cout << *r << endl;
    return 0;
}  // main
//  Results:
// Enter a number: 17
// 17
// 17
// 17
```

19

Pointers and Functions

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

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

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

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



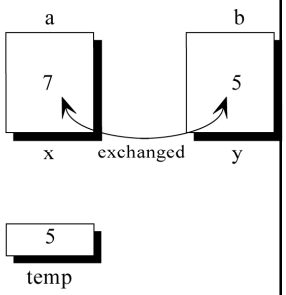
(a) Original values unchanged

20

Pointers and Functions - 2

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

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

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

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



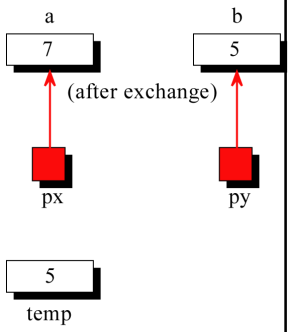
(b) Original values exchanged

21

Alternative to pass by reference – pass a pointer by value and use it to change the original variable.

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

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

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

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



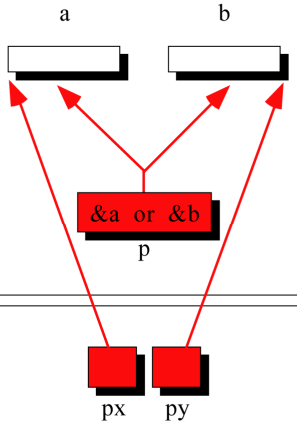
(c) Original values exchanged

22

Functions, returning pointers

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

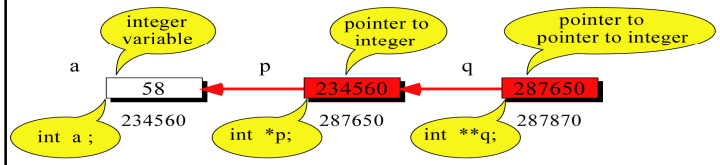
```
int   *smaller  (int  *px,
                 int  *py )
{
  return  ( *px < *py ? px : py ) ;
}  // smaller
```



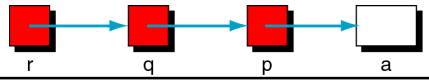
23

Pointers to pointers

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




```
//  Statements
a = 58 ;
p = &a ;
q = &p ;
cout <<        a << " ";
cout <<       *p << " ";
cout <<      **q << " ";
```



- there is no limit on how many levels of indirection can be used.

24

```
// P09-06 Using pointers to pointers
#include <iostream.h>
int main (void)
{ //Local Declarations
    int   a;          int  *p;   int  **q; int ***r;
// Statements
    p = &a;           q = &p;   r = &q;
    cout << "Enter a number: "; cin  >> a;
    cout << "Your number is: " << a << endl;
    cout << "\nEnter a number: "; cin  >> *p;
    cout << "Your number is: " << a << endl;
    cout << "\nEnter a number: "; cin  >> **q;
    cout << "Your number is: " << a << endl;
    cout << "\nEnter a number: "; cin  >> ***r;
    cout << "Your number is: " << a << endl;
    return 0;
}  // main
/*        Results:
    Enter a number: 1
    Your number is: 1
    Enter a number: 2
    Your number is: 2
    Enter a number: 3
    Your number is: 3
    Enter a number: 4
    Your number is: 4
*/
```

25

Compatibility

In addition to its own attributes, pointers take on attributes of the type to which it refers.

Assign a pointer of one type to a different type – illegal.

```
// P09-07  Demonstrate size of pointers
#include <iostream.h>
int main (void)
{ //       Local Declarations
    char  c; char *pc;
    int   sizeofc          = sizeof(c);
    int   sizeofpc         = sizeof(pc);
    int   sizeofStarpc     = sizeof(*pc);

    int  a; int *pa;
    int  sizeofa           = sizeof(a);
    int  sizeofpa          = sizeof(pa);
    int  sizeofStarpa      = sizeof(*pa);

    double  x; double *px;
    int    sizeofx         = sizeof(x);
    int    sizeofpx        = sizeof(px);
    int    sizeofStarpx    = sizeof(*px);
```

26

```
// Statements
    cout << "sizeof(c): "   << sizeofc      << " | ";
    cout << "sizeof(pc): "  << sizeofpc     << " | ";
    cout << "sizeof(*pc): " << sizeofStarpc << endl;

    cout << "sizeof(a): "   << sizeofa      << " | ";
    cout << "sizeof(pa): "  << sizeofpa     << " | ";
    cout << "sizeof(*pa): " << sizeofStarpa << endl;
    cout << "sizeof(x): "   << sizeofx      << " | ";
    cout << "sizeof(px): "  << sizeofpx     << " | ";
    cout << "sizeof(*px): " << sizeofStarpx << endl;
    return 0;
}  // 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
*/
```

27

Void pointer can be used with any type, but *cannot be dereferenced.*

void *pVoid ;

Casting:

int a ; char *p ;

p = (char*) &a ;

Valid, but dangerous assignments:

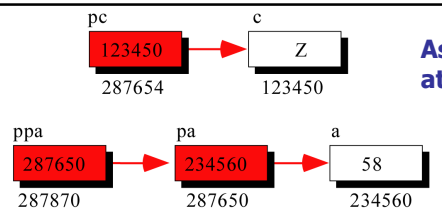
void *pVoid ; char *pChar ; int *pInt ;

pVoid = pChar ;

pInt = pVoid ;

pInt = (int*) pChar ;

28

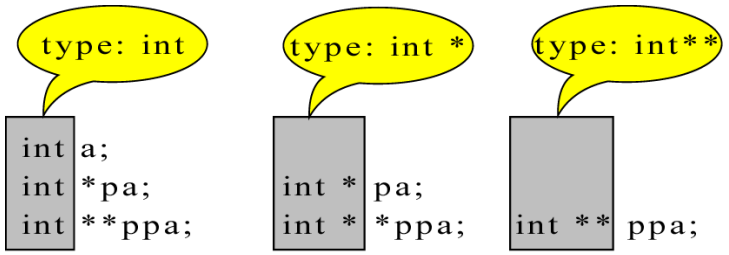


Assignment must be at the correct level

```
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
```

29

Pointer types must match

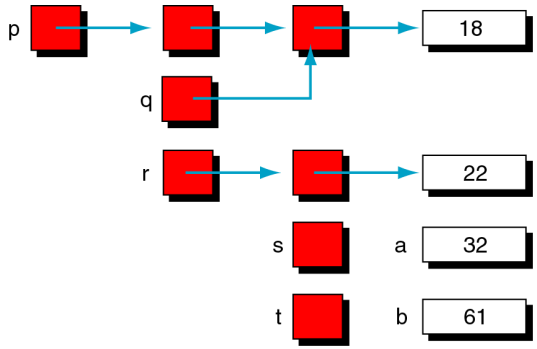



```
a      = 4;
*pa    = 4;      pa     = &a;
**ppa  = 4;      *ppa   = &a;      ppa = &pa;
```

30

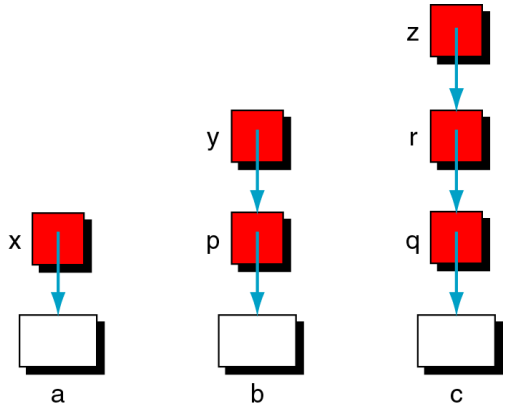
Assuming all variables are integer, and all pointers are of proper type,
show the final values of the variables in the figure after:

```
a=***p;
s=**p;
t=*p;
b=**r;
**q=b;
```



31

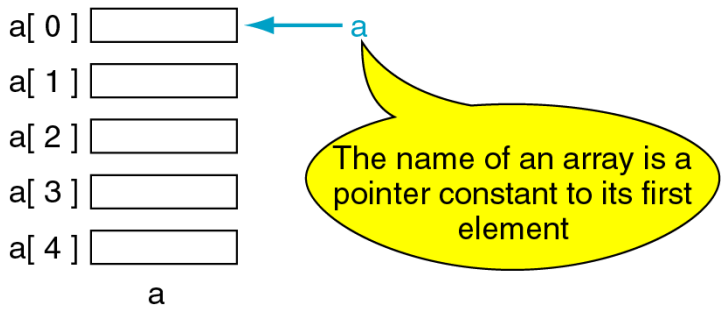
- Write a program that creates the structure shown in the figure and reads data into **a** and **b**, using pointers **x** and **y**. The program then *multiplies* the value of **a** by **b** and stores the *result* in **c** *using pointers **x**, **y** and **z**.* Finally, all three variable are printed using the pointers **x**, **y**, and **z**.



32

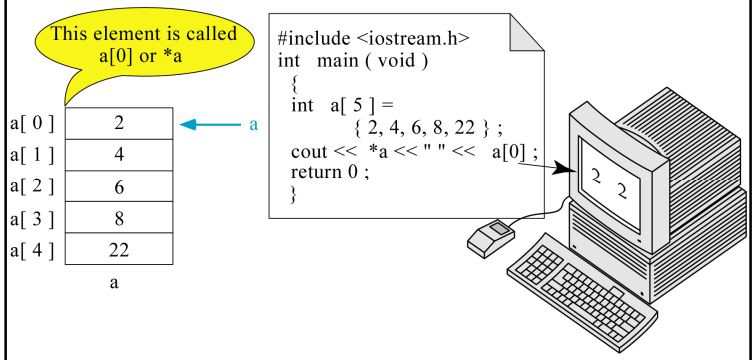
Arrays and Pointers

- *Array name is a const pointer to the first element.*
- *Dereference array name* access first element *a[0].*



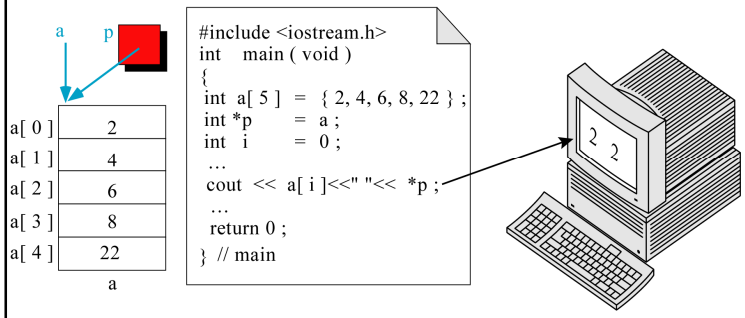
33

Dereference of array name



34

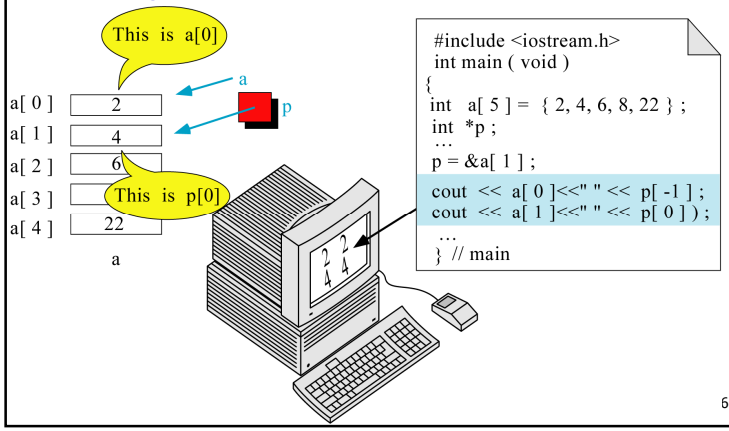
Array names as pointers



35

Multiple Array Pointers

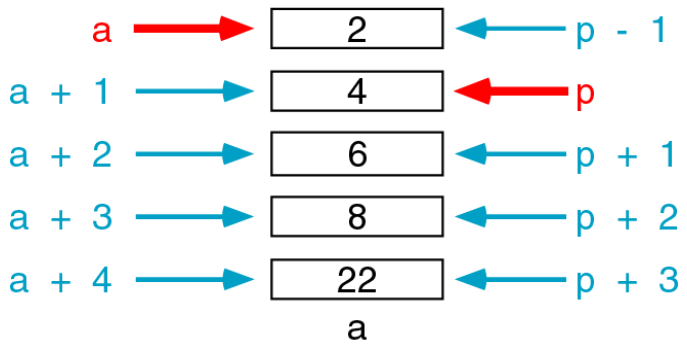
- *Pointers can be used with indexes* to access array elements.
- When a pointer is not referencing the first element, it *can have negative offset.*



6

Pointer Arithmetic and Arrays

- Adding integer ***n*** to a pointer moves pointer to an ***nth*** element (the size of the element is determined by the type of the pointer).



37

address = pointer + (offset * size_of_element)

Basic operations:

Addition – pointer and integer.

Subtraction – two pointers, or pointer and integer.

Subtraction returns an index - the **number of elements** between 2 pointers ***(not the number of bytes).***

p+5 5+p p-5 p1-p2 p++ --p

Relational operators – *only* for pointers of the *same type*.

p1>=p2 p1 != p2

if (ptr != NULL)

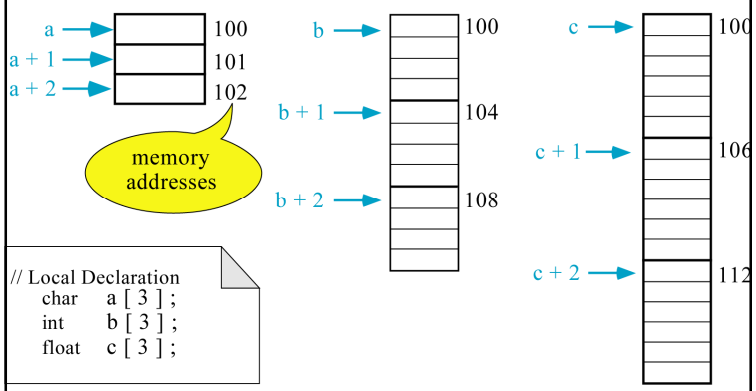
if (ptr == NULL)

38

Pointer arithmetic and different types

address = pointer + (offset * sizeof element)

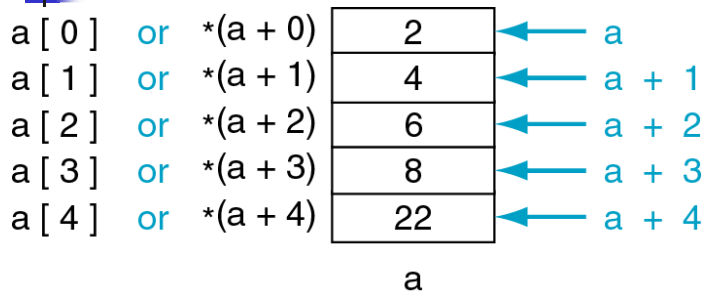
a+n → a+n*(sizeof(one element))



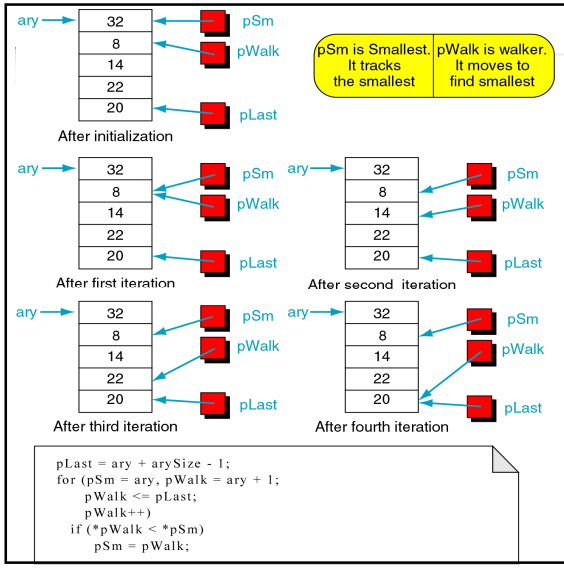

```
// Local Declaration
    char    a [ 3 ] ;
    int     b [ 3 ] ;
    float   c [ 3 ] ;
```

39

Dereferencing array pointers



40



Find the smallest element using pointers

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

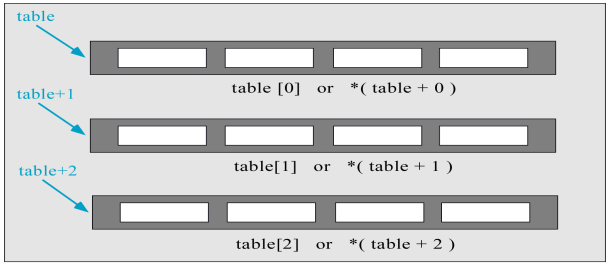
41

```
/* Print an array forward by adding 1 to a pointer. Then print it backward
   by subtracting one    */
#include <stdio.h>
#define MAX_SIZE 10
int main (void)
{  int  ary[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
   int *pWalk;
   int *pEnd;
   printf("Array forward : ");
   for (pWalk = ary, pEnd = ary + MAX_SIZE; pWalk <  pEnd; pWalk++)
      printf ("%3d", *pWalk);
   printf ("\n");
   printf("Array backward: ");
   for (pWalk = pEnd - 1; pWalk >= ary; pWalk--)
      printf ("%3d", *pWalk);
   printf ("\n");
}  /* main */
/* Results:
   Array forward :   1  2  3  4  5  6  7  8  9 10
   Array backward:  10  9  8  7  6  5  4  3  2  1
*/
```

2

Pointers and 2D arrays

- Array name is a *const* pointer to the *first row of the array*.
- *Dereference of array name* is a *pointer* to a *1D array.*



int table[3][4] ;

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

Print table

43

Passing an array to a function

Function call:

f(aryName) ;

Function prototype:

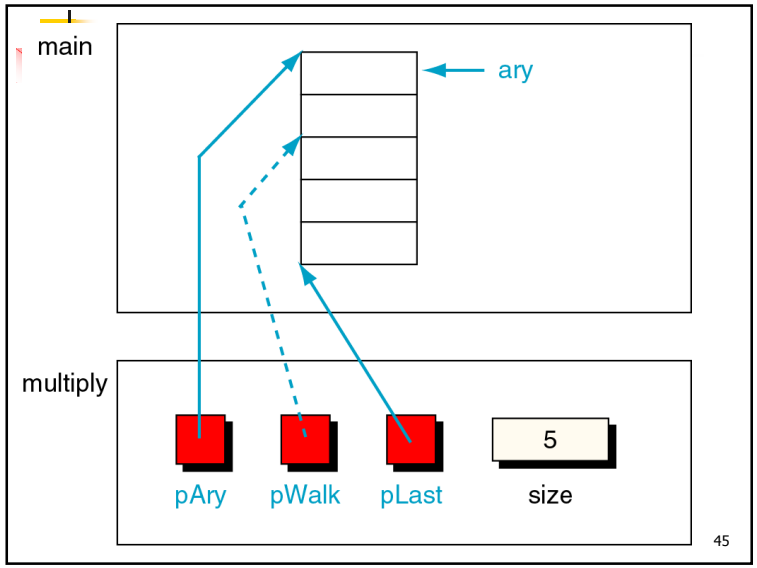
int f(int ary[]) ; // as array – const pointer

// as pointer – masks data structure, more effective ,

// needs good descriptive name

int f(int *arySalary) ;

44



45

```
// P09-09 Multiply array elements by two
#include <iostream.h>
#include <iomanip.h>
const short SIZE = 5;
//  Prototype Declarations
void multiply (int* pAry, int size);
int main (void)
{ //      Local Declarations
    int  ary [SIZE];        int* pLast;        int* pWalk;
//  Statements
    pLast = ary + SIZE - 1;
    for (pWalk = ary; pWalk <= pLast; pWalk++)
       {
        cout << "Please enter an integer: ";
        cin  >> *pWalk;
       } // for pWalk
    multiply (ary, SIZE);
    cout << "Doubled size is: \n";
    for (pWalk = ary; pWalk <= pLast; pWalk++)
       cout << setw(4) << *pWalk << endl;
    return 0;
}  // main
```

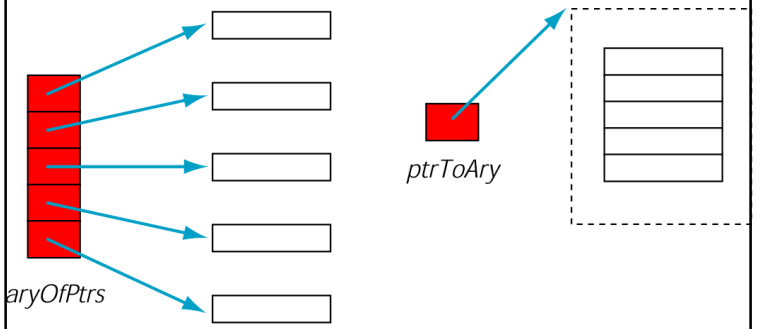
46

```
/*  ==== multiply ====
    Multiply elements in an array by 2
    Pre    pAry is pointer to full array
           size indicates number of elements in array
    Post   Values in array doubled.
*/
void multiply (int* pAry, int  size)
{
//  Local Declarations
    int* pWalk;        int* pLast;

//  Statements
    pLast = pAry + size - 1;
    for (pWalk = pAry; pWalk <= pLast; pWalk++)
       *pWalk = *pWalk * 2;
    return;
}   // multiply
/*  Results:
    Please enter an integer: 2
    Please enter an integer: 4
    Please enter an integer: 6
    Please enter an integer: 8
    Please enter an integer: 10
    Doubled size is:
      4
      8
     12
     16
     20
*/
```

47

Array of pointers versus pointer to array

```
int *aryOfPts[5] ;
int (*ptrToAry)[5] ;
```



(a) An array of pointers (b) A pointer to an array

48

Show what would be printed from the following block:

```
{
 int num[5] = { 3, 4, 6, 2, 1 };
 int *p = num;
 int *q = num +2;
 int *r  = &num[1];
 printf("\n%d %d", num[2], *(num + 2));
 printf("\n%d %d", *p, *(p + 1));
 printf("\n%d %d", *q, *(q + 1));
 printf("\n%d %d", *r, *(r + 1));
 return 0;
}
```

49

Show what would be printed from the following block:

```
{
/* Local Definitions */
   int x [2][3] =
     {
      { 4  , 5 , 2 } ,
      { 7 ,  6 , 9 }
     };
   int (*p) [3]  = &x [1];
   int (*q) [3]  =  x;
/* Statements */
   printf("\n%d %d %d", (*p)[0],(*p)[1],(*p)[2]);
    printf("\n%d %d",    *q[0],  *q[1]);
}
```

50

Given the following definitions:

```
int num[26] = {23, 3, 5, 7, 4, -1, 6};
int *n = num;
int i  = 2;
int j  = 4;
```

show the value of the following expressions:

a. `n`
b. `*n`
c. `*n + 1`
d. `*(n + 1)`
e. `*n + j`
f. `*&i`

51

Given the following definitions:

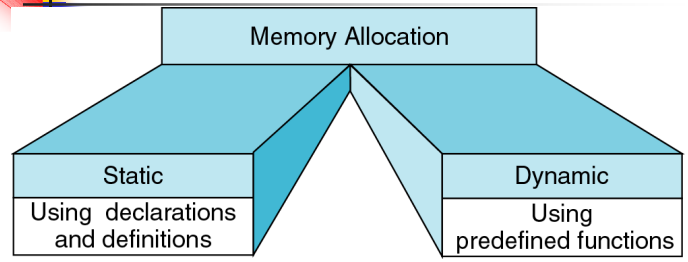
```
int data[15] = {5,2,3,4,1,3,7,2,4,3,2,9,12};
```

show the value of the following expressions:

a. `data + 4`
b. `*(data + 4)`
c. `*data + 4`
d. `*(data + (*data + 2))`

52

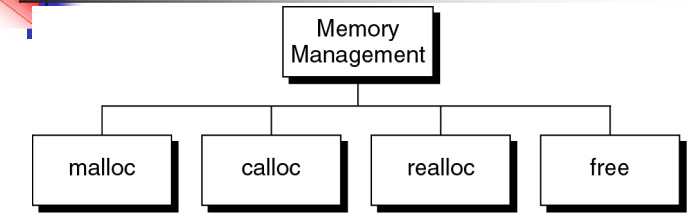
Memory allocation functions



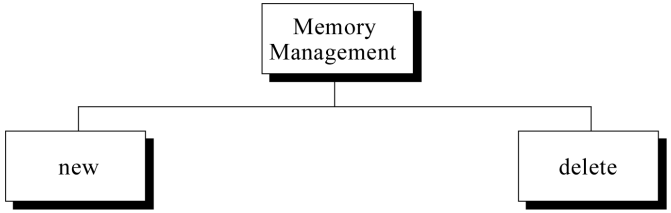
- *main()* – must be in memory all the time
- each called function needs to be in memory only while active
- the code for a function can be in memory all the time
- the *local variables* are available only when the *function is active*
- it is *possible for more then 1 version of a function to be active at a time – (recursion) multiple copies of the local variables will be allocated (stack memory) , only 1 copy of a function.*
- *Heap memory* – assigned dynamically during program execution.

53

C

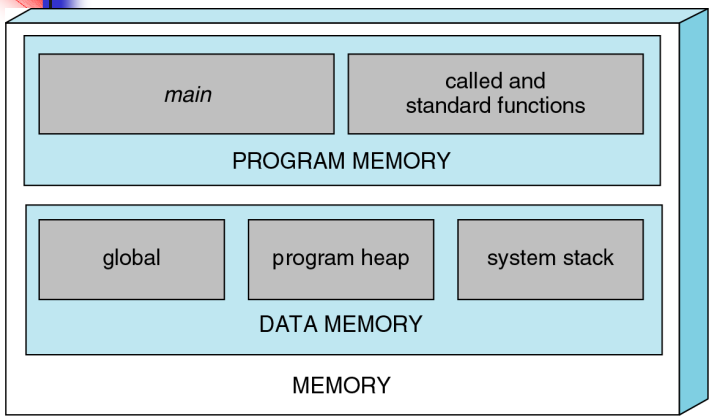


C++



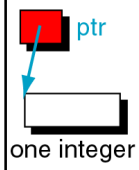
54

A Conceptual View of Memory



55

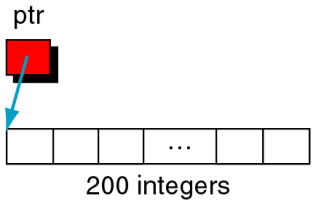
Memory allocation




```
if (!(ptr = (int *)malloc(sizeof(int))))
  /* No memory available */
  exit (100) ;
/* Memory available */
...
```

56

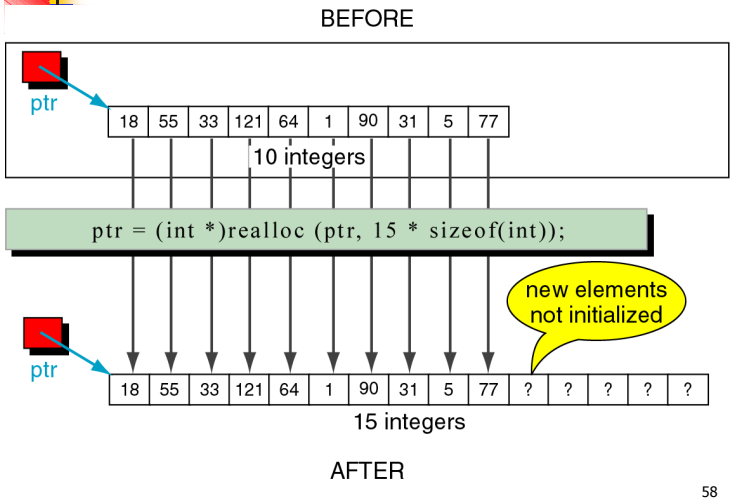
Memory allocation for an array




```
if (!(ptr = (int *)calloc (200, sizeof(int))))
    /* No memory available */
    exit (100) ;

/* Memory available */
...
```

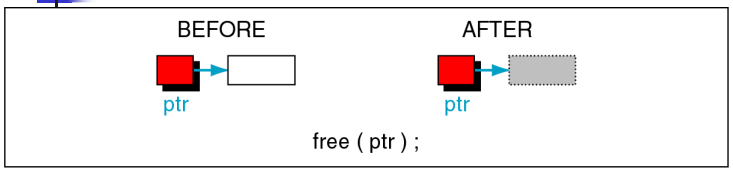
57

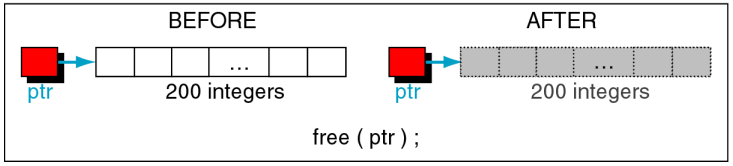
Change array size



58

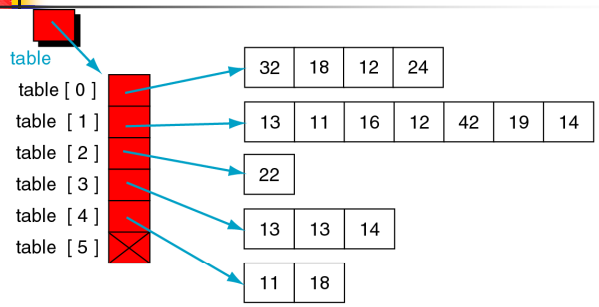
Freeing memory





59

Array of pointers



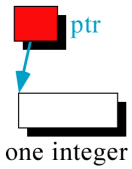

```
table = (int **)calloc (rowNum + 1, sizeof(int*));

table[0] = (int*)calloc (4, sizeof(int));
table[1] = (int*)calloc (7, sizeof(int));
table[2] = (int*)calloc (1, sizeof(int));
table[3] = (int*)calloc (3, sizeof(int));
table[4] = (int*)calloc (2, sizeof(int));
table[5] = NULL;
```

60

new memory allocation

initializing:

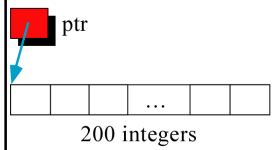
ptr = new int(40) ;




```
int  *ptr;
if ( ! (ptr = new int ))
      //  No memory available
      exit (100) ;

//  Memory available
    …
```

61

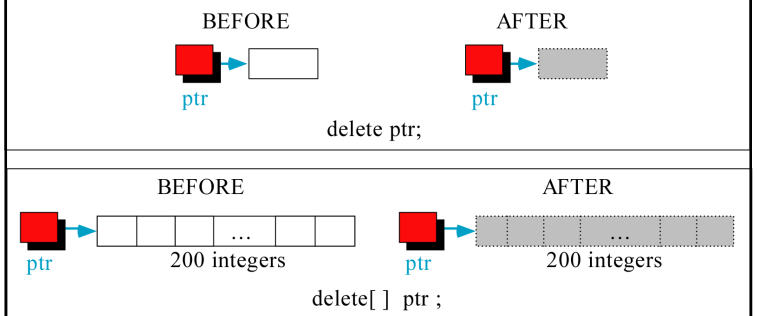
Memory allocation for an array




```
if ( ! ( ptr = new  int[200] ))
      //  No memory available
      exit (100) ;

//  Memory available
…
```

62

Releasing Memory



- *Release does not change the value in the pointer.*
- It is a *logical error* to use the pointer after memory was released.
- It is an *error to delete* memory that was *not* allocated with *new.*

63

Allocating space for 2D array

int **ptr ;

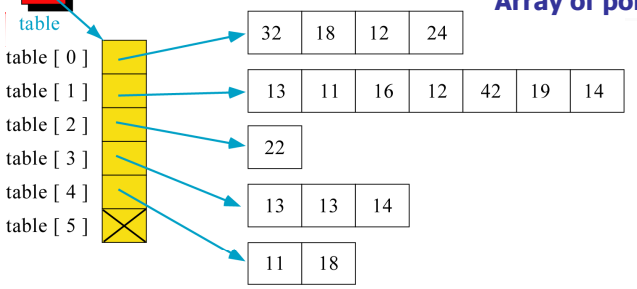
ptr = new int[numItems][12] ;

delete [] ptr ;

- OS releases all memory upon program termination.

64

Array of pointers




```
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 ;
```

65

```
//   P09-20 Testing memory reuse
#include <iostream.h>
int main (void)
{ //Declarations
    int looper;        char* ptr;
//   Statements
    for (looper = 0; looper < 5; looper++)
            {
             ptr = new char[16];
             cout << "Memory allocated at: " << &ptr << endl;

             delete[] ptr;
            } // for
    return 0;
}   // main
/*  Results: You will get different answers.
    Results in Personal Computer:
            Memory allocated at: 0x00f341b6
            Memory allocated at: 0x00f341b6
            Memory allocated at: 0x00f341b6
            Memory allocated at: 0x00f341b6
            Memory allocated at: 0x00f341b6
    Results in UNIX system:
            Memory allocated at: 10001010
            Memory allocated at: 10001010
            Memory allocated at: 10001010
            Memory allocated at: 10001010
            Memory allocated at: 10001010
*/
```

66