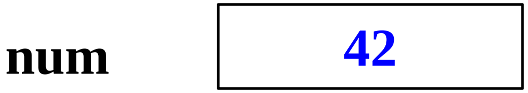
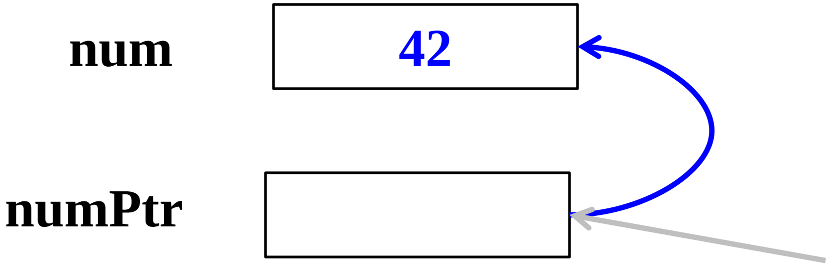
Pointers
Introduction

What is a pointer?

- Simple variables: An int / float variable is like a box which can store a single int value such as 42.



- A pointer does not store a simple value directly. Instead, a pointer stores a reference to another value.



A pointer variable. The current value is a reference to the pointee num above.

Introduction

- A pointer is a variable that represents the location (rather than the value) of a data item.

- A pointer is just another kind of value
- Pointer type declaration:

```
int *numPtr;
float *fp;
```

Basic Concept

- Every stored data item occupies one or more contiguous memory cells depending on its type (char, int, double, etc.).

- Whenever we declare a variable, the system allocates memory location(s) to hold the value of the variable.

- This location has a unique address.

Contd.

- Consider the statement

```
int   xyz = 50;
```

  – The compiler will allocate a location for the integer variable **xyz**, and put the value 50 in that location.

  – Suppose that the address location chosen is 1380.

xyz	**variable**
50	**value**
1380	**address**

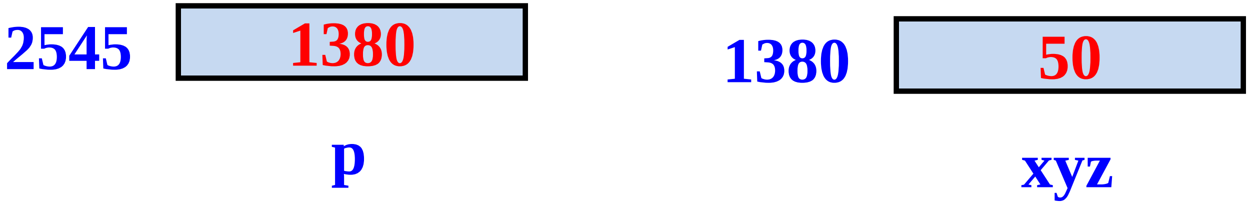
Contd.

- Suppose we assign the address of xyz to a variable p.

```
int *p;
p = &xyz;
```

p is said to point to the variable xyz.

Variable	Value	Address
xyz	50	1380
p	1380	2545



2545 1380

p

1380 50

xyz

Accessing the Address of a Variable

- The address of a variable can be determined using the ‘&’ operator.

```
p = &xyz;
```

- The ‘&’ operator can be used only with a simple variable or an array element.

```
        &distance
        &x[0]
        &x[i-2]
```

Contd.

- The following usages are illegal:

```
&235              Pointing at constant.


int  arr[20];
  :
&arr              Pointing at array name.

&(a+b)            Pointing at expression.
```

Example

```
#include  <stdio.h>
int main( ) {
   int   a;
   float  b, c;
   double  d;
   char  ch;

    a = 10;   b = 2.5;  c = 12.36;  d = 12345.66;  ch = ‘A’;
   printf  (“%d is stored in location %u \n”,  a,  &a) ;
   printf  (“%f is stored in location %u \n”,  b,  &b) ;
   printf  (“%f is stored in location %u \n”,  c,  &c) ;
   printf  (“%ld is stored in location %u \n”,  d,  &d) ;
   printf  (“%c is stored in location %u \n”,  ch,  &ch) ;
}
```

Output:

```
10 is stored in location 3221224908              a
2.500000 is stored in location 3221224904        b
12.360000 is stored in location 3221224900       c
12345.660000 is stored in location 3221224892    d
A is stored in location 3221224891              ch
```

Pointer Declarations

- Pointer variables must be declared before we use them.
- General form:

```
data_type   *pointer_name;
```

```
int* ptr_a,  ptr_b;
```

ptr_a is of type pointer to int, ptr_b is an int!

```
int* ptr_a,  *ptr_b;
```

ptr_a and ptr_b are of type pointer to int.

Contd.

- Once a pointer variable has been declared, it can be made to point to a variable using an assignment statement .

```
int     *ip;
float   *fp;
int    count;
float  speed;

:
ip = &count;
fp = &speed;
```

  – This is called **pointer initialization**.

Pointer Operations in C

- Creation
  - & variable Returns variable’s memory address
- Dereference
  - * pointer Returns contents stored at address
- Indirect assignment
  - *pointer = val Stores value at address
- Assignment
  - pointer = ptr Stores pointer in another variable

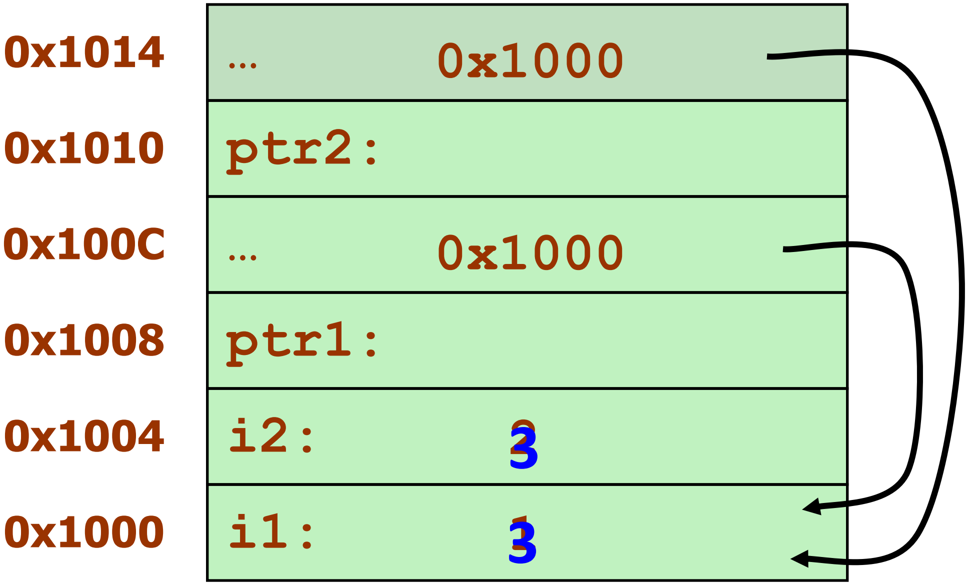
Using Pointers

```
int  i1;
int  i2;
int *ptr1;
int *ptr2;

i1 = 1;
i2 = 2;

ptr1 = &i1;
ptr2 = ptr1;

*ptr1 = 3;
i2 = *ptr2;
```



0x1014	… 0x1000
0x1010	ptr2:
0x100C	… 0x1000
0x1008	ptr1:
0x1004	i2: 2 3
0x1000	i1: 1 3

Using Pointers (cont.)

```
int  int1     = 1036;   /* some data to point to  */
int  int2     = 8;

int *int_ptr1 = &int1;  /* get addresses of data  */
int *int_ptr2 = &int2;

*int_ptr1 = int_ptr2;

*int_ptr1 = int2;
```

What happens?

Type check warning: `int_ptr2` **is not an** `int`

`int1` **becomes 8**

Using Pointers (cont.)

```
int  int1     = 1036;   /* some data to point to  */
int  int2     = 8;

int *int_ptr1 = &int1;  /* get addresses of data  */
int *int_ptr2 = &int2;

int_ptr1 = *int_ptr2;

int_ptr1 = int_ptr2;
```

What happens?

Type check warning: `*int_ptr2` **is not an** `int *`

Changes `int_ptr1` **– doesn't change** `int1`

Example Code

```
int x = 1, y = 2, z[10];
int *ip;                  // ip is a pointer to an int

ip = &x;                  // ip now points to where x is stored
y = *ip;                  // set y equal to the value pointed to by
                           // ip, or y = x
*ip = 0;                  // change the value that ip points to to
                          //0, so now x=0, but  y is unchanged
ip = &z[0];           // now ip points at the 0th location in array z

*ip = *ip + 1;          // (z[0]) is incremented
```

Pointer Arithmetic

pointer + number **pointer – number**

E.g., pointer + 1 adds 1 something to a pointer

```
char   *p;
char    a;
char    b;

p = &a;
p += 1;
```

Pointer arithmetic should be used cautiously

In each, p now points to b

(Assuming compiler doesn't reorder variables in memory)

```
int   *p;
int    a;
int    b;

p = &a;
p += 1;
```

Adds 1*sizeof(char) to the memory address

Adds 1*sizeof(int) to the memory address

Scale Factor

- We have seen that an integer value can be added to or subtracted from a pointer variable.

```
int   *p1,  *p2 ;
int   i,  j;
:
p1  =  p1  +  1 ;
p2  =  p1  +  j ;
p2++ ;
p2  =  p2  –  (i + j) ;
```

- In reality, it is not the integer value which is added/subtracted, but rather the scale factor times the value.

Contd.

Data Type	Scale Factor = sizeof (data type)
char	1
int	4
float	4
double	8

– If p1 is an integer pointer, then

p1++

will increment the value of p1 by 4.

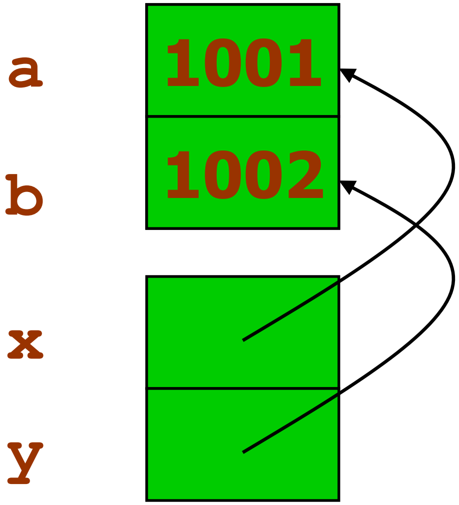
Passing Pointers to a Function

- Pointers are often passed to a function as arguments.
  - Allows data items within the calling program to be accessed by the function, altered, and then returned to the calling program in altered form.

Pass-by-Reference

```
void set_x_and_y(int *x,   int *y) {
  *x = 1001;
  *y = 1002;
}

void f(void) {
  int a = 1;
  int b = 2;
  set_x_and_y( &a,&b);
}
```



Example: passing arguments by value

```
#include  <stdio.h>
int main() {
     int  a, b;
     a = 5 ;   b = 20 ;
     swap (a, b) ;
     printf  (“\n a = %d,  b = %d”, a, b);
}

void   swap  (int  x, int  y) {
     int  t ;
     t = x ;
     x = y ;
     y = t ;
}
```

a and b do not swap

x and y swap

Output

a = 5, b = 20

Example: passing arguments by passing the reference

```
#include  <stdio.h>
int main() {
     int  a, b;
     a = 5 ;   b = 20 ;
     swap (&a, &b) ;
     printf  (“\n a = %d,  b = %d”, a, b);
}

void   swap  (int  *x, int  *y) {
     int  t ;
     t = *x ;
     *x = *y ;
     *y = t ;
}
```

*(&a) and *(&b) swap

*x and *y swap

Output

a = 20, b = 5

scanf Revisited

```
int   x,  y ;
printf  (“%d %d %d”,  x, y, x+y) ;
```

- What about scanf ?

```
scanf   (“%d %d %d”, x, y, x+y) ;        NO

scanf   (“%d %d”, &x, &y) ;              YES
```

Example: Sort 3 integers

- Three-step algorithm:
  1. Read in three integers x, y and z
  2. Put smallest in x
     - Swap x, y if necessary; then swap x, z if necessary.
  3. Put second smallest in y
     - Swap y, z if necessary.

sort3 as a function

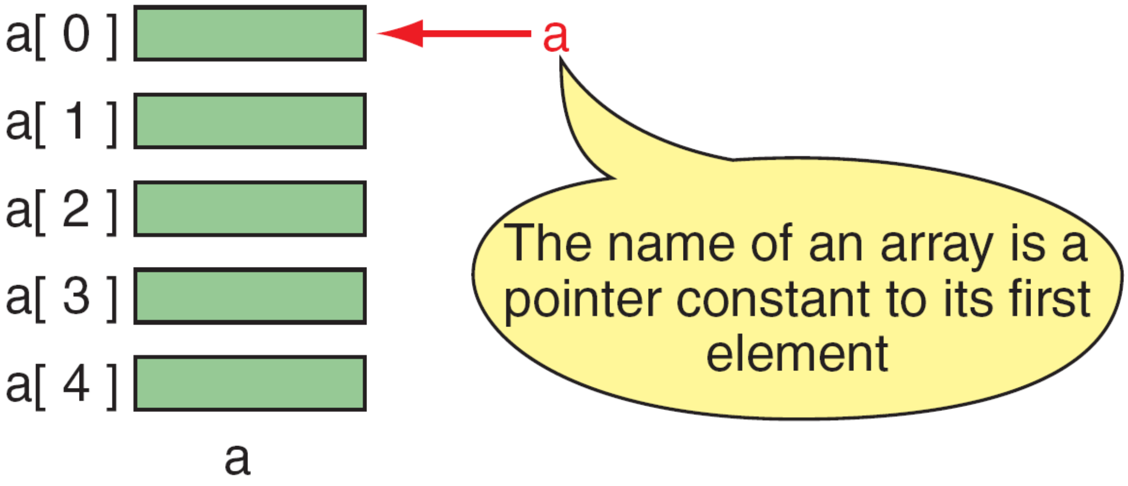
```
int main() {
     int  x, y, z ;
     ………..
     scanf  (“%d %d %d”, &x, &y, &z) ;
     sort3  (&x, &y, &z) ;
     ………..
}

void   sort3  (int *xp,  int *yp,  int *zp)
{
     if  (*xp > *yp)   swap (xp, yp);
     if  (*xp > *zp)   swap (xp, zp);
     if  (*yp > *zp)   swap (yp, zp);
}
```

xp/yp/zp are pointers

Pointers and Arrays

- When an array is declared,
  - The compiler allocates a base address and sufficient amount of storage to contain all the elements of the array in contiguous memory locations.
  - The base address is the location of the first element (index 0) of the array.
  - The compiler also defines the array name as a constant pointer to the first element (element 0).



The name of an array is a pointer constant to its first element

Note

$$a \xleftrightarrow{\text{same}} \&a[0]$$

a is a pointer only to the first element—not the whole array.

The name of an array is a pointer constant;
it cannot be used as an *lvalue*.

Example

- Consider the declaration:

```
int  a[5]  =  {1, 2, 3, 4, 5} ;
```

Type of a is int *

- – Suppose that the base address of a is 2500, and each integer requires 4 bytes.

Element	Value	Address
a[0]	1	2500
a[1]	2	2504
a[2]	3	2508
a[3]	4	2512
a[4]	5	2516

Contd.

x ⇔ &a[0] ⇔ 2500 ;

- **p = a;** and **p = &a[0];** are equivalent.
- We can access successive values of x by using **p++** or **p--** to move from one element to another.

- Relationship between p and x:

```
p    =  &a[0]  =  2500
p+1  =  &a[1]  =  2504
p+2  =  &a[2]  =  2508
. . .
```

***(p+i) gives the value of x[i]**

Arrays and Pointers

int a[5] = { 1, 2, 3, 4, 5 } ;

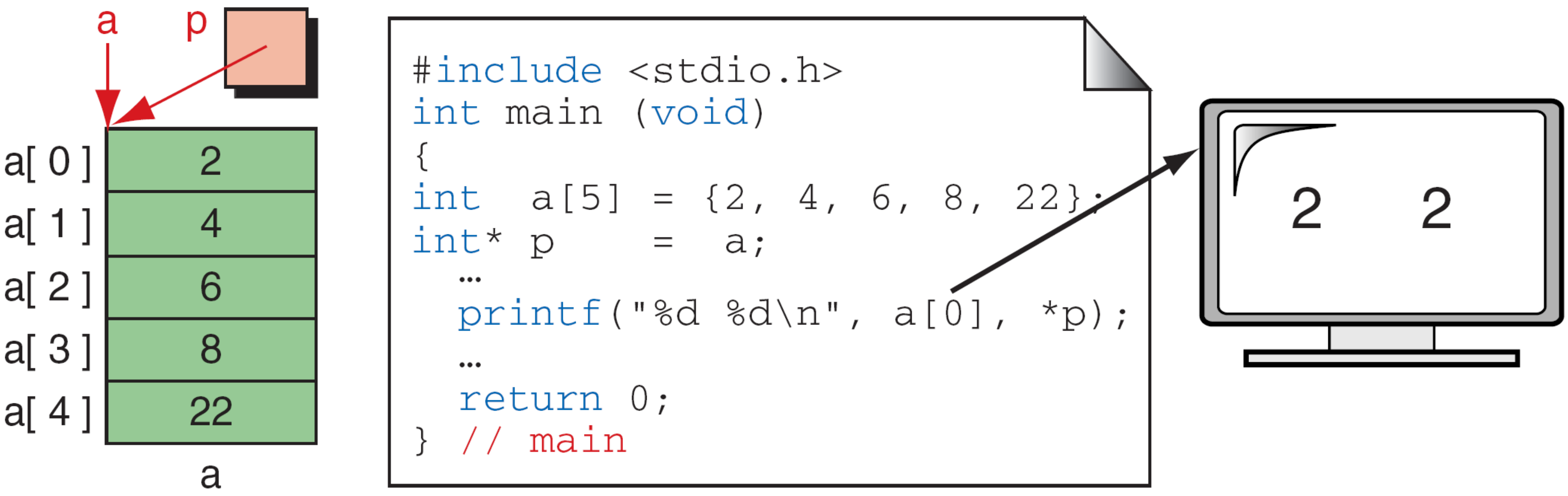
`int *p; int i, j;`

- Let **`p = A;`**
- Then **`p`** points to **`A[0]`**

  `p + i` points to **`A[i]`**

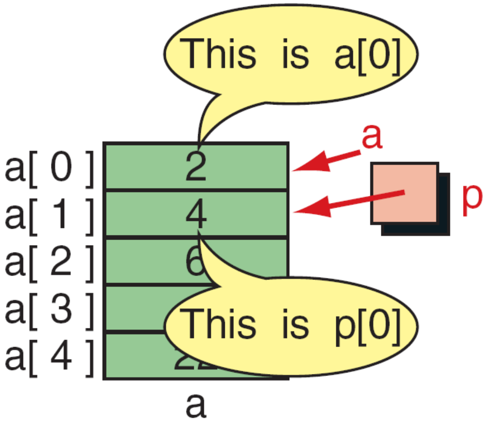
  `&A[j] == p+j`

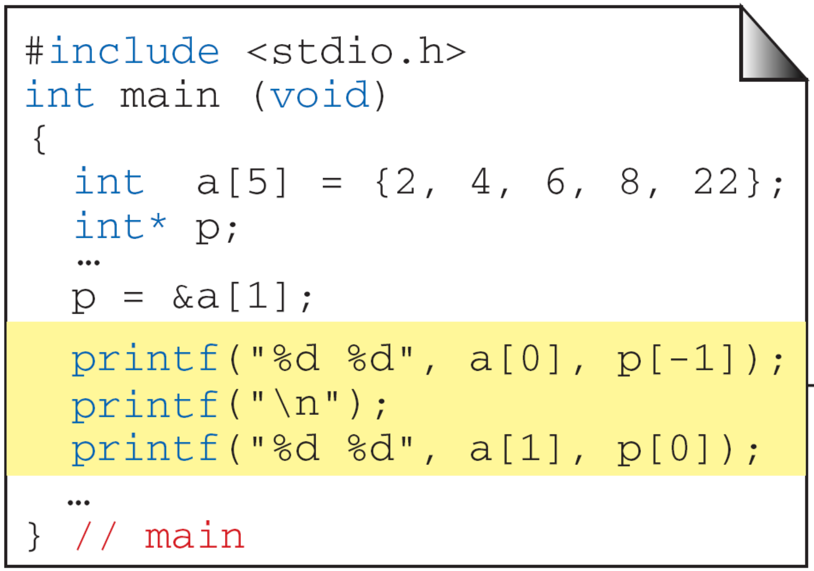
  `*(p+j)` is the same as **`A[j]`**



```
#include <stdio.h>
int main (void)
{
int  a[5] = {2, 4, 6, 8, 22};
int* p    =  a;
  …
  printf("%d %d\n", a[0], *p);
  …
  return 0;
} // main
```

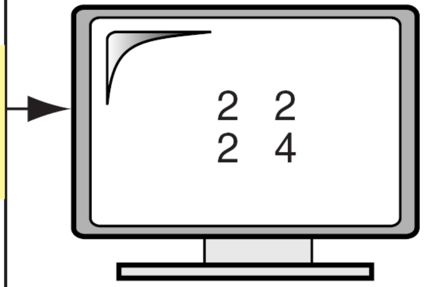
```
2   2
```





```
#include <stdio.h>
int main (void)
{
   int  a[5] = {2, 4, 6, 8, 22};
   int* p;
   …
   p = &a[1];

   printf("%d %d", a[0], p[-1]);
   printf("\n");
   printf("%d %d", a[1], p[0]);
   …
} // main
```



```
2  2
2  4
```

Note

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

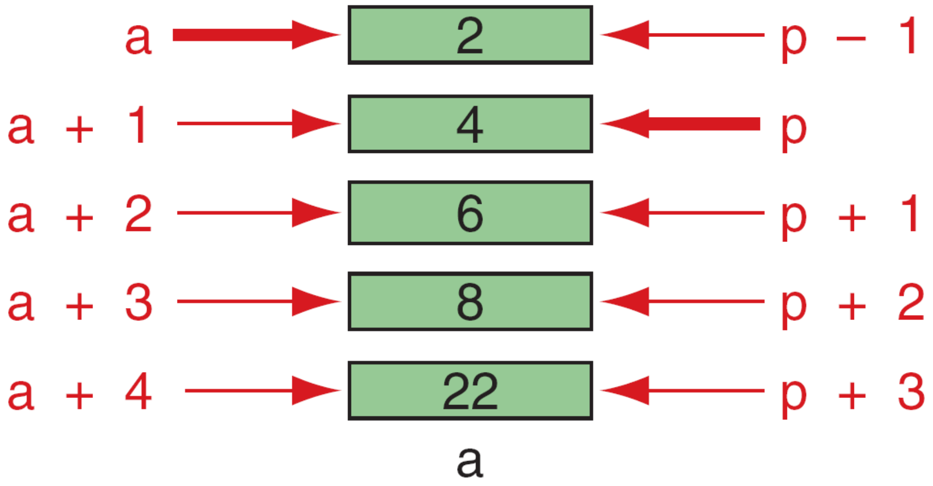


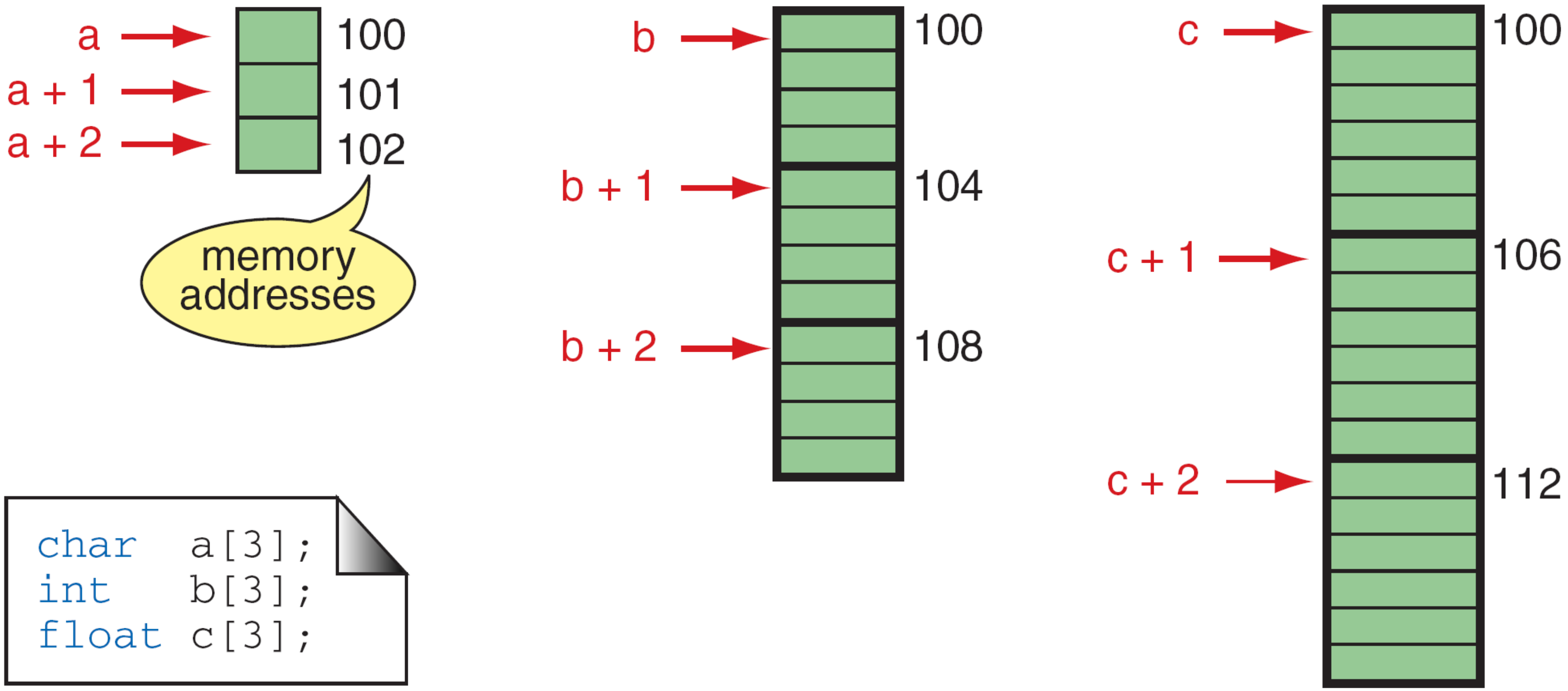
FIGURE 10-5 Pointer Arithmetic

Note

a + n

↓

a + n * (sizeof (one element))



```
char  a[3];
int   b[3];
float c[3];
```

FIGURE 10-6 Pointer Arithmetic and Different Types

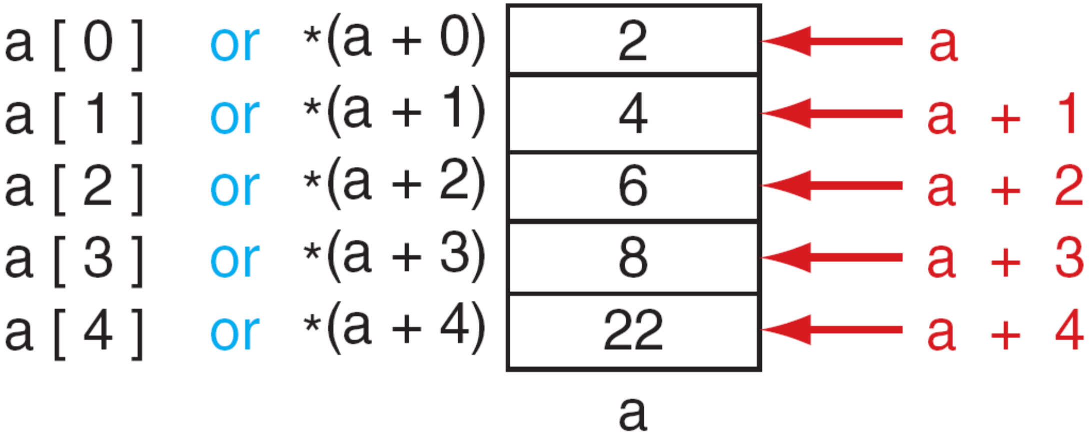


FIGURE 10-7 Dereferencing Array Pointers

Note

The following expressions are identical.

***(a + n)   and   a[n]**

Arrays and Pointers

Array ≈ pointer to the initial (0th) array element

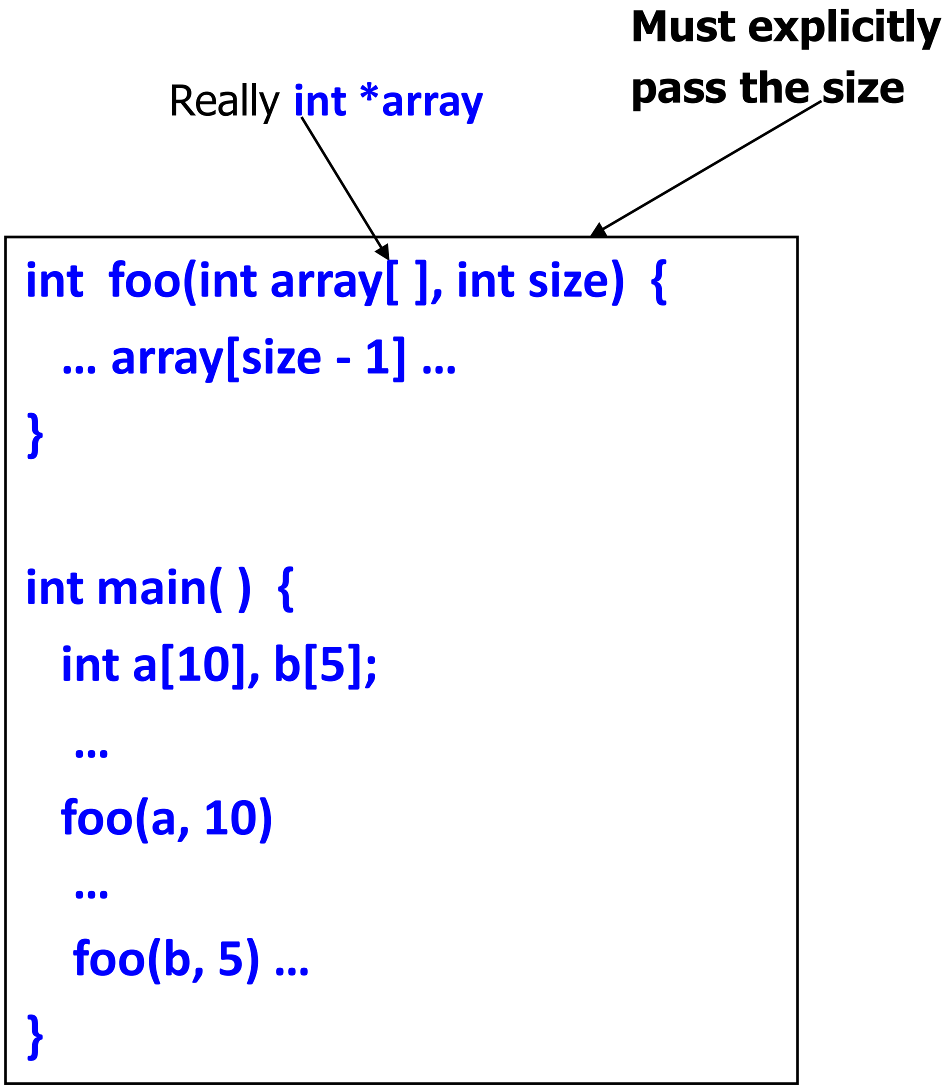
$a[i] \equiv *(a+i)$

An array is passed to a function as a pointer

int foo(int array[ ], int size) and

int foo(int *array, int size) are identical.

Passing arrays:



Really **int *array**

Must explicitly pass the size

```
int  foo(int array[ ], int size)  {
   … array[size - 1] …
}


int main( )  {
   int a[10], b[5];
    …
   foo(a, 10)
    …
    foo(b, 5) …
}
```

Arrays and Pointers

```
int  foo(int array[],  int size) {
  …
  printf(“%d\n”, sizeof(array));
}

int  main(void) {
   int a[10], b[5];
   …
  foo(a, 10)
  …
  foo(b, 5) …
  printf(“%d\n”, sizeof(a));
}
```

What does this print?

8

… because `array` is really a pointer

What does this print?

40

Example: function to find average

```
int main()
{
    int  x[100], k, n ;

    scanf  (“%d”, &n) ;

    for  (k=0; k<n; k++)
        scanf  (“%d”, &x[k]) ;

    printf  (“\nAverage is %f”,
                          avg (x, n) );
}
```

int *array

```
float  avg  (int array[ ], int  size)
{
    int  *p, i , sum = 0;

    p = array ;

    for  (i=0; i<size; i++)
        sum = sum + *(p+i);

    return  ((float) sum / size);
}
```

p[i]