Pointers as Arguments

- Example:

```
void decompose(double x,
     long *int_part, double *frac_part)
{
      *int_part = (long) x;
      *frac_part = x - *int_part;
}
```

- Arguments in calls of scanf are pointers:

```
int i;
…
scanf("%d", &i);
```

Another example:

```
int i, *p;
. . .
p = &i;
scanf("%d", p);
scanf("%d", &p);   /*** WRONG ***/
```

- Failing to pass a pointer to a function when one is expected can have disastrous results.

A call of decompose in which & operator is missing:

```
decompose(3.14159, i, d);
```

When decompose stores values in *int_part and *frac_part, it will attempt to change unknown memory locations instead of modifying i and d.

If we've provided a prototype for decompose, the compiler will detect the error.

- In the case of scanf, however, failing to pass pointers may go undetected.

Using const to Protect Arguments

- const goes in the parameter's declaration, just before the specification of its type:

```
void f(const int *p) {
  *p = 0;   /*** WRONG ***/
}
```

- Attempting to modify *p is an error that the compiler will detect.

Pointers as Return Values

- Functions are allowed to return pointers:

```
int *max(int *a, int *b) {
  if (*a > *b)
    return a;
  else
    return b;
}
```

- A function could also return a pointer to an external variable or to a static local variable.
- Never return a pointer to an *automatic* local variable:

```
int *f(void) {
  int i;
  …
  return &i;
}
```

The variable i won't exist after f returns.

- Pointers can point to array elements.
- If a is an array, then &a[i] is a pointer to element i of a.
- It's sometimes useful for a function to return a pointer to one of the elements in an array.
- A function that returns a pointer to the middle element of a, assuming that a has n elements:

```
int *find_middle(int a[], int n) {
  return &a[n/2];
}
```

Chapter 12 Pointers and Arrays

- C allows us to perform arithmetic—addition and subtraction—on pointers to array elements: ***pointer arithmetic*** or ***address arithmetic***.

This leads to an alternative way of processing arrays in which pointers take the place of array subscripts.

- Example:

```
int a[10]={0}, *p, *q;
p = &a[2];
*p = 3 ;
q = p+3 ;//Add an integer to a pointer
*q = 6;
*(p-1)=2;
*(q-5)=1;
```

- Caution: when performing ***pointer arithmetic,*** make sure the pointer do not point to an address out of the bounds of an array.

Comparing Pointers

- Pointers can be compared using the relational operators (<, <=, >, >=) and the equality operators (== and !=).

Using relational operators is meaningful only for pointers to elements of the same array.

- The outcome of the comparison depends on the relative positions of the two elements in the array.

  Example: After the assignments

```
p = &a[5];
q = &a[1];
```

  the value of p <= q is 0 and the value of p >= q is 1.

Pointers to Compound Literals (C99)

- It's legal for a pointer to point to an element within an array created by a compound literal:

```
int *p = (int []){3, 0, 3, 4, 1};
```

Using Pointers for Array Processing

- Pointer arithmetic allows us to visit the elements of an array by repeatedly incrementing a pointer variable.

```
#define N 10
…
int a[N], sum, *p;
…
sum = 0;
for (p = &a[0]; p < &a[N]; p++)
  sum += *p;
```

- The condition p < &a[N] in the for statement deserves special mention.

Combining the * and ++ Operators

Expression	*Meaning*
*p++ or *(p++)	Value of expression is *p before increment; increment p later.
(*p)++	Value of expression is *p before increment; increment *p later.
*++p or *(++p)	Increment p first; value of expression is *p after increment
++*p or ++(*p)	Increment *p first; value of expression is *p after increment

Using an Array Name as a Pointer

- Pointer arithmetic is one way in which arrays and pointers are related.
- Another key relationship:

  The name of an array can be used as a pointer to the first element in the array.

- Example:

```
int a[10];
*a = 7;      /* stores 7 in a[0] */
*(a+1) = 12; /* stores 12 in a[1] */
```

- Although an array name can be used as a pointer, it's not possible to assign it a new value.

```
while(*a != 0)
    a++;           /*** WRONG ***/
```

- How about the code segment

```
int a[10], *p;
p = a;
while(*p != 0)
    p++;
```

Array Arguments (Revisited)

- When passed to a function, an array name is treated as a pointer. Example:

```
int find_largest(int a[], int n) {
  int i, max;

  max = a[0];
  for (i = 1; i < n; i++)
    if (a[i] > max)
      max = a[i];
  return max;
}
```

- A call of find_largest:

```
largest = find_largest(b, N);
```

  This call causes a pointer to the first element of b to be assigned to a; the array itself isn't copied.

- The fact that an array argument is treated as a pointer has some important consequences.
- ***Consequence 1:*** When an ordinary variable is passed to a function, its value is copied; any changes to the corresponding parameter don't affect the variable.

  In contrast, an array used as an argument isn't protected against change.

  To indicate that an array parameter won't be changed, we can include the word const in its declaration.

- ***Consequence 2:*** The time required to pass an array to a function doesn't depend on the size of the array.

  There's no penalty for passing a large array, since no copy of the array is made.

- ***Consequence 3:*** An array parameter can be declared as a pointer if desired.
- Example:

```
int find_largest(int *a, int n)
{
 …
}
```

- Declaring a to be a pointer is equivalent to declaring it to be an array; the compiler treats the declarations as though they were identical.
- Although declaring a *parameter* to be an array is the same as declaring it to be a pointer, the same isn't true for a *variable*.

  The following declaration causes the compiler to set aside space for 10 integers:

```
int a[10];
```

  The following declaration causes the compiler to allocate space for a pointer variable:

```
int *a;
```

- In the latter case, a is not an array; attempting to use it as an array can have disastrous results.

  For example, the assignment

```
*a = 0;   /*** WRONG ***/
```

  is an error, since we don't know where a is pointing, the effect on the program is undefined.

- *Consequence 4:* A function with an array parameter can be passed an array "slice"—a sequence of consecutive elements.

  An example that applies find_largest to elements 5 through 14 of an array b:

```
largest = find_largest(&b[5], 10);
```

Using a Pointer as an Array Name

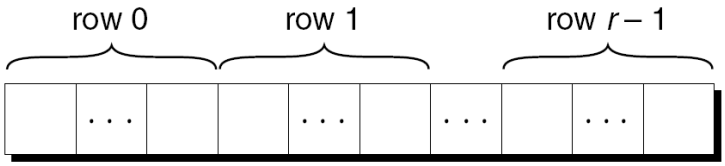
- C allows us to subscript a pointer as though it were an array name:

```
#define N 10
. . .
int a[N], i, sum = 0, *p = a;
. . .
 for (i = 0; i < N; i++)
   sum += p[i];
```

  The compiler treats p[i] as *(p+i).

Pointers and Multidimensional Arrays

- Just as pointers can point to elements of one-dimensional arrays, they can also point to elements of multidimensional arrays.
- Chapter 8 showed that C stores two-dimensional arrays in row-major order.
- Layout of an array with *r* rows:



- If p initially points to the element in row 0, column 0, we can visit every element in the array by incrementing p repeatedly.
- Consider the problem of initializing all elements of the following array to zero:

```
int a[10][11];
```

- The obvious technique would be to use nested for loops:

```
int row, col;
. . .
for (row = 0; row < 10; row++)
     for (col = 0; col < 11; col++)
          a[row][col] = 0;
```

- If we view a as a one-dimensional array of integers, a single loop is sufficient:

```
int *p;
. . .
for(p=&a[0][0]; p<=&a[10-1][11-1]; p++)
  *p = 0;
```

- This example also shows that C treats a two-dimensional array as one-dimensional.

Processing the Rows of a Multidimensional Array

- A pointer variable p can also be used for processing the elements in just one *row* of a two-dimensional array.
- To visit the elements of row i, we'd initialize p to point to element 0 in row i in the array a:

```
p = &a[i][0];
```

  or we could simply write

```
p = a[i];
```

- The line above means that for any two-dimensional array a, the expression a[i] is a pointer to the first element in row i.
- A loop that clears row i of the array a:

```
int a[NUM_ROWS][NUM_COLS], *p, i;
  . . .
for(p=a[i]; p<a[i]+NUM_COLS; p++)
    *p = 0;
```

- Since a[i] is a pointer to row i of the array a, we can pass a[i] to a function that's expecting a one-dimensional array as its argument.
- In other words, a function that's designed to work with one-dimensional arrays will also work with a row belonging to a two-dimensional array.
- But, processing the elements in a *column* of a two-dimensional array isn't as easy, because arrays are stored by row, not by column.
- A loop that clears column i of the array a:

```
int a[NUM_ROWS][NUM_COLS],
    (*p)[NUM_COLS], i;
  . . .
for(p=&a[0]; p<&a[NUM_ROWS]; p++)
    (*p)[i] = 0;
```

- The name of *any* array can be used as a pointer, regardless of how many dimensions it has, but some care is required.
- Example:

```
int a[NUM_ROWS][NUM_COLS];
```

  a is **not** a pointer to `a[0][0]`, but a pointer to `a[0]`.

  C regards `a` as a one-dimensional array whose elements are one-dimensional arrays.

  When used as a pointer, a has type `int (*)[NUM_COLS]` (pointer to an integer array of length NUM_COLS).
- Knowing that a points to a[0] is useful for simplifying loops that process the elements of a two-dimensional array.
- Instead of writing

```
for(p=&a[0]; p<&a[NUM_ROWS]; p++)
  (*p)[i] = 0;
```

- to clear column i of the array a, we can write

```
for(p=a; p<a+NUM_ROWS; p++)
  (*p)[i] = 0;
```

- We can "trick" a function into thinking that a multidimensional array is really one-dimensional.
- Example: If function has prototype

```
int find_largest(int *a, int n);
```

  The following code is wrong:

```
#define NUM_ROWS  11
#define NUM_COLS  12
int m, a[NUM_ROWS][NUM_COLS];
 . . .
m=find_largest(a, NUM_ROWS*NUM_COLS);
```

- This an error, because the type of `a` is

```
int (*)[NUM_COLS]
```

  but find_largest is expecting type `int *`.
- The correct call is

```
m=find_largest(a[0],NUM_ROWS*NUM_COLS);
```

  a[0] points to element 0 in row 0, and it has type int * (after conversion by the compiler).

Pointers and Variable-Length Arrays (C99)

- Pointers are allowed to point to elements of variable-length arrays (VLAs).
- Example:

```
void f(int n)
{
  int a[n], *p;
  p = a;
  …
}
```

- When the VLA has more than one dimension, the type of the pointer depends on the length of each dimension except for the first.

  A two-dimensional example:

```
void f(int m, int n)
{
  int a[m][n], (*p)[n];
  p = a;
  …
}
```

- Since the type of p depends on n, which isn't constant, p is said to have a ***variably modified type.***

- **A potential problem:**

  The validity of an assignment such as p = a can't always be determined by the compiler.

  The following code will compile but is correct only if m and n are equal:

```
int a[m][n], (*p)[m];
p = a;
```

  If m is not equal to n, any subsequent use of p will cause undefined behavior.

Pointer arithmetic works with VLAs.

- A two-dimensional VLA:

```
int a[m][n];
```

- A pointer capable of pointing to a row of a:

```
int (*p)[n];
```

- A loop that clears column i of a:

```
for(p = a; p < a + m; p++)
   (*p)[i] = 0;
```

- Restriction on variable-length arrays:

  the declaration must be inside the body of a function or in a function prototype.