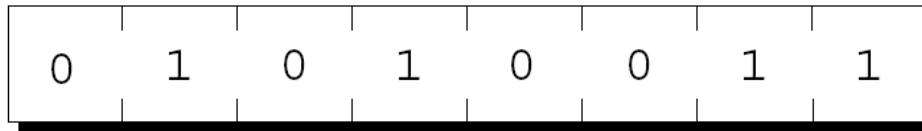


Chapter 11

Pointers

Pointer Variables

- The first step in understanding pointers is visualizing what they represent at the **machine level (i.e., low level or memory level)**.
- In most modern computers, main memory is divided into **bytes**, with each byte capable of storing eight bits of information:



- Each byte has a unique **address**.

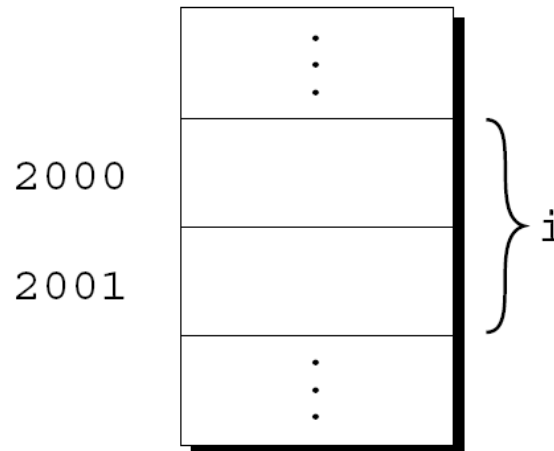
Pointer Variables

- If there are n bytes in memory, we can think of addresses as numbers that range from 0 to $n - 1$:

Address	Contents
0	01010011
1	01110101
2	01110011
3	01100001
4	01101110
	⋮
$n-1$	01000011

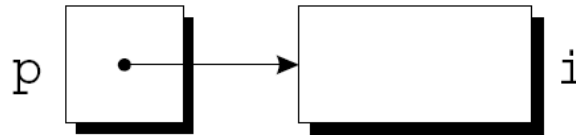
Pointer Variables

- Each variable in a program occupies one or more bytes of memory.
- The address of the first byte is said to be the address of the variable.
- In the following figure, the address of the variable `i` is 2000:



Pointer Variables

- Addresses can be stored in special ***pointer variables***.
- When we store the address of a variable ***i*** in the pointer variable ***p***, we say that ***p*** “points to” ***i***.
- A graphical representation:



Declaring Pointer Variables

- When a pointer variable is declared, its name must be preceded by an asterisk:

```
int *p;
```

- `p` is a pointer variable capable of pointing to *objects* of type `int`.
- We use the term *object* instead of *variable* since `p` might point to an area of memory that doesn't belong to a variable.

Declaring Pointer Variables

- Pointer variables can appear in declarations along with other variables:

```
int i, j, a[10], b[20], *p, *q;
```

- C requires that every pointer variable point only to objects of a particular type (the ***referenced type***):

```
int *p;      /* points only to integers */
double *q;   /* points only to doubles */
char *r;     /* points only to characters */
```

- There are no restrictions on what the referenced type may be.

The Address and Indirection Operators

- C provides a pair of operators designed specifically for use with pointers.
 - To find the address of a variable, we use the **&** (**address**) operator. **& is ampersand**
 - To gain access to the object that a pointer points to, we use the ***** (**indirection**) operator.

The Address Operator

- Declaring a pointer variable sets aside space for a pointer but doesn't make it point to an object:
`int *p; /* points nowhere in particular */`
- It's crucial to initialize `p` before we use it.

The Address Operator

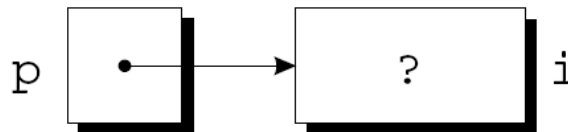
- One way to initialize a pointer variable is to assign it the address of a variable:

```
int i, *p;
```

```
...
```

```
p = &i;
```

- Assigning the address of `i` to the variable `p` makes `p` point to `i`:



The Address Operator

- It's also possible to initialize a pointer variable at the time it's declared:

```
int i;  
int *p = &i;
```

- The declaration of `i` can even be combined with the declaration of `p`:

```
int i, *p = &i;
```

The Indirection Operator

- Once a pointer variable points to an object, we can use the `*` (indirection) operator to access what's stored in the object.
- If `p` points to `i`, we can print the value of `i` as follows:

```
printf("%d\n", *p);
```

- Applying `&` to a variable produces a pointer to the variable. Applying `*` to the pointer takes us back to the original variable:

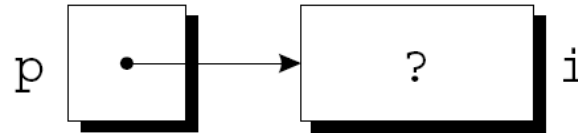
```
j = *&i;    /* same as j = i; */
```

The Indirection Operator

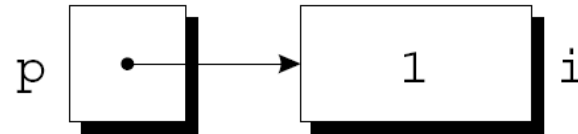
- As long as `p` points to `i`, `*p` is an *alias* for `i`.
 - `*p` has the same value as `i`.
 - Changing the value of `*p` changes the value of `i`.
- The example on the next slide illustrates the equivalence of `*p` and `i`.

The Indirection Operator

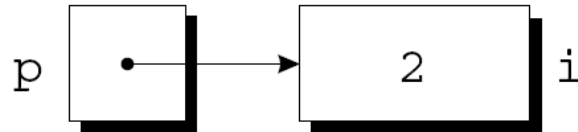
```
p = &i;
```



```
i = 1;
```



```
printf("%d\n", i);      /* prints 1 */  
printf("%d\n", *p);    /* prints 1 */  
*p = 2;
```




```
printf("%d\n", i);      /* prints 2 */  
printf("%d\n", *p);    /* prints 2 */
```

The Indirection Operator

- Applying the indirection operator to an uninitialized pointer variable causes undefined behavior:

```
int *p;  
printf("%d", *p);    /* ** WRONG ** */
```



- Assigning a value to `*p` is particularly dangerous:

```
int *p;  
*p = 1;    /* ** WRONG ** */
```

Pointer Assignment

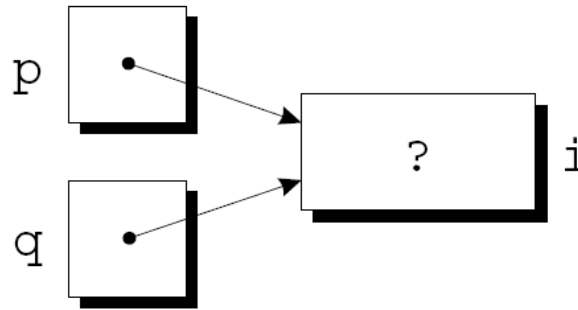
- C allows the use of the assignment operator to copy pointers of the same type.
- Assume that the following declaration is in effect:
`int i, j, *p, *q;`
- Example of pointer assignment:
`p = &i;`

Pointer Assignment

- Another example of pointer assignment:

`q = p;`

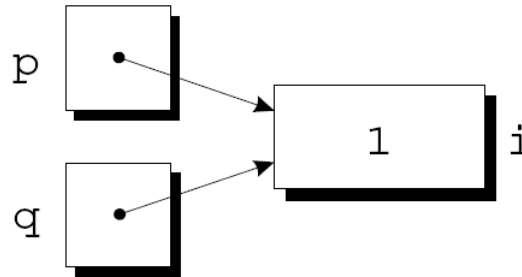
q now points to the same place as p:



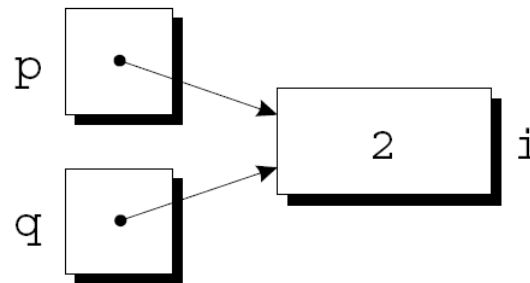
Pointer Assignment

- If `p` and `q` both point to `i`, we can change `i` by assigning a new value to either `*p` or `*q`:

`*p = 1;`



`*q = 2;`



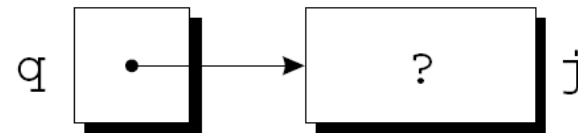
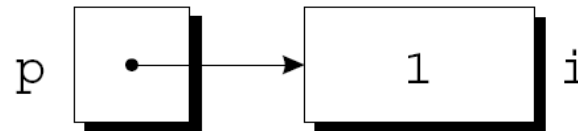
- Any number of pointer variables may point to the same object.

Pointer Assignment

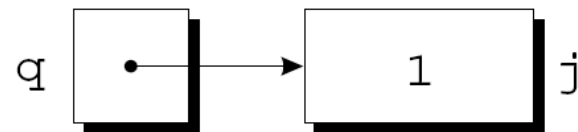
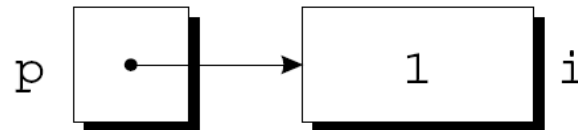
- Be careful not to confuse
`q = p;`
with
`*q = *p;`
- The first statement is a pointer assignment, but the second is not.
- The example on the next slide shows the effect of the second statement.

Pointer Assignment

```
p = &i;  
q = &j;  
i = 1;
```



```
*q = *p;
```



Pointers as Arguments

- In Chapter 9, we tried—and failed—to write a `decompose` function that could modify its arguments.
- By passing a *pointer* to a variable instead of the *value* of the variable, `decompose` can be fixed.

Pointers as Arguments

- New definition of decompose:

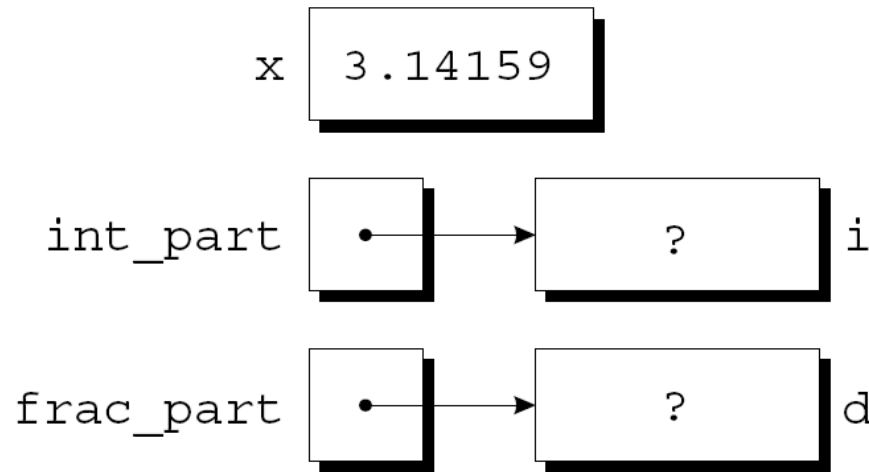
```
void decompose(double x, long *int_part,  
               double *frac_part)  
{  
    *int_part = (long) x;  
    *frac_part = x - *int_part;  
}
```

- Possible prototypes for decompose:

```
void decompose(double x, long *int_part,  
               double *frac_part);  
void decompose(double, long *, double *);
```

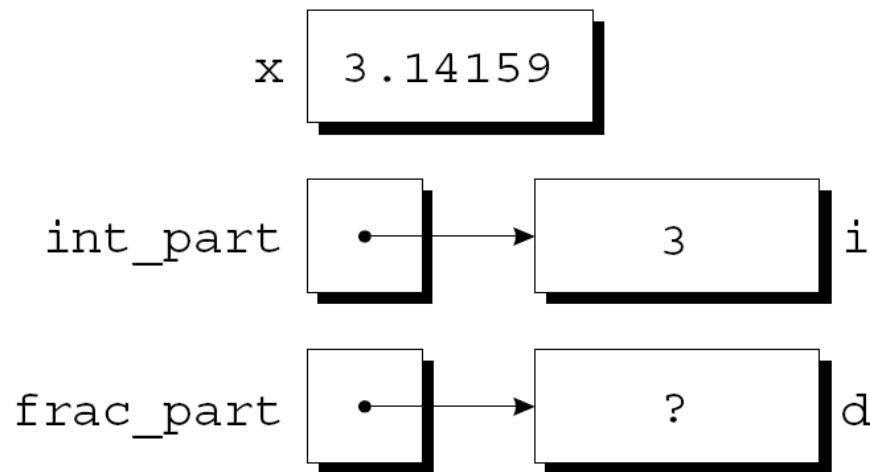
Pointers as Arguments

- A call of `decompose`:
`decompose(3.14159, &i, &d);`
- As a result of the call, `int_part` points to `i` and `frac_part` points to `d`:



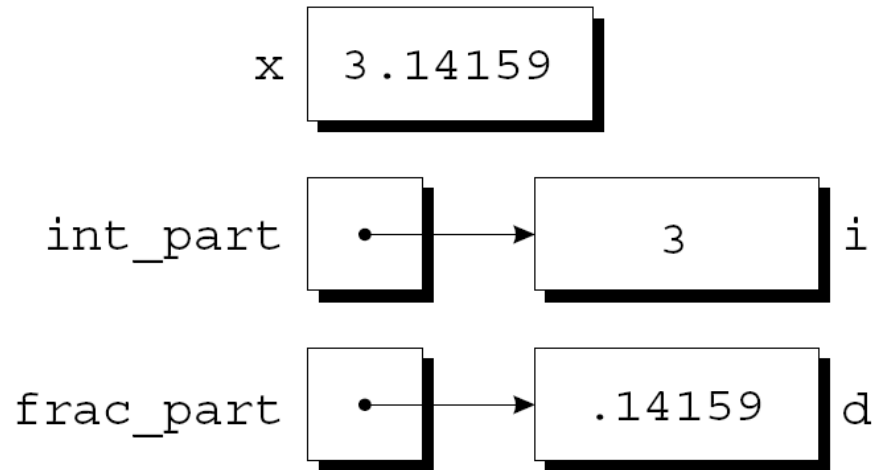
Pointers as Arguments

- The first assignment in the body of `decompose` converts the value of `x` to type `long` and stores it in the object pointed to by `int_part`:



Pointers as Arguments

- The second assignment stores $x - *int_part$ into the object that `frac_part` points to:



Pointers as Arguments

- Arguments in calls of `scanf` are pointers:

```
int i;
```

```
...
```

```
scanf("%d", &i);
```

Without the `&`, `scanf` would be supplied with the *value* of `i`.

Pointers as Arguments

- Although `scanf`'s arguments must be pointers, it's not always true that every argument needs the `&` operator:

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

- Using the `&` operator in the call would be wrong:

```
scanf("%d", &p);    /* ** WRONG ** */
```

Pointers as Arguments

- Failing to pass a pointer to a function when one is expected can have disastrous results.
- A call of `decompose` in which the `&` 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.

Program: Finding the Largest and Smallest Elements in an Array

- The `max_min.c` program uses a function named `max_min` to find the largest and smallest elements in an array.
- Prototype for `max_min`:

```
void max_min(int a[], int n, int *max, int *min);
```
- Example call of `max_min`:

```
max_min(b, N, &big, &small);
```
- When `max_min` finds the largest element in `b`, it stores the value in `big` by assigning it to `*max`.
- `max_min` stores the smallest element of `b` in `small` by assigning it to `*min`.

Program: Finding the Largest and Smallest Elements in an Array

- `max_min.c` will read 10 numbers into an array, pass it to the `max_min` function, and print the results:

Enter 10 numbers: 34 82 49 102 7 94 23 11 50 31

Largest: 102

Smallest: 7

maxmin.c

```
/* Finds the largest and smallest elements in an array
 */

#include <stdio.h>

#define N 10

void max_min(int a[], int n, int *max, int *min);

int main(void)
{
    int b[N], i, big, small;

    printf("Enter %d numbers: ", N);
    for (i = 0; i < N; i++)
        scanf("%d", &b[i]);
}
```

Chapter 11: Pointers

```
    max_min(b, N, &big, &small);

    printf("Largest: %d\n", big);
    printf("Smallest: %d\n", small);

    return 0;
}

void max_min(int a[], int n, int *max, int *min)
{
    int i;

    *max = *min = a[0];
    for (i = 1; i < n; i++) {
        if (a[i] > *max)
            *max = a[i];
        else if (a[i] < *min)
            *min = a[i];
    }
}
```


Using **const** to Protect Arguments

- When an argument is a pointer to a variable `x`, we normally assume that `x` will be modified:
`f(&x);`
- It's possible, though, that `f` merely needs to examine the value of `x`, not change it.
- The reason for the pointer might be efficiency: passing the value of a variable can waste time and space if the variable requires a large amount of storage.

Using **const** to Protect Arguments

- We can use `const` to document that a function won't change an object whose address is passed to the function.
- `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 call of the max function:

```
int *p, i, j;
...
p = max(&i, &j);
```

After the call, *p* points to either *i* or *j*.

Pointers as Return Values

- Although `max` returns one of the pointers passed to it as an argument, that's not the only possibility.
- 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 as Return Values

- 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];  
}
```