11 Pointers

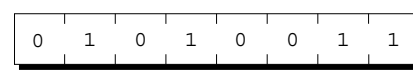
The 11th commandment was "Thou Shalt Compute"
or "Thou Shalt Not Compute"—I forget which.

Pointers are one of C's most important—and most often misunderstood—features. Because of their importance, we'll devote three chapters to pointers. In this chapter, we'll concentrate on the basics; Chapters 12 and 17 cover more advanced uses of pointers.

We'll start with a discussion of memory addresses and their relationship to pointer variables (Section 11.1). Section 11.2 then introduces the address and indirection operators. Section 11.3 covers pointer assignment. Section 11.4 explains how to pass pointers to functions, while Section 11.5 discusses returning pointers from functions.

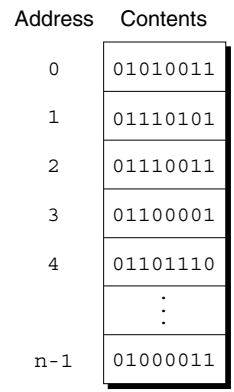
11.1 Pointer Variables

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

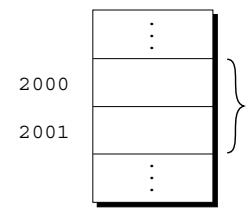


Each byte has a unique ***address*** to distinguish it from the other bytes in memory. If there are n bytes in memory, we can think of addresses as numbers that range from 0 to $n - 1$ (see the figure at the top of the next page).

An executable program consists of both code (machine instructions corresponding to statements in the original C program) and data (variables in the original program). Each variable in the program occupies one or more bytes of memory;



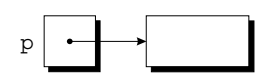
the address of the first byte is said to be the address of the variable. In the following figure, the variable `i` occupies the bytes at addresses 2000 and 2001, so `i`'s address is 2000:



Here's where pointers come in. Although addresses are represented by numbers, their range of values may differ from that of integers, so we can't necessarily store them in ordinary integer variables. We can, however, store them 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`. In other words, a pointer is nothing more than an address, and a pointer variable is just a variable that can store an address.

Q&A

Instead of showing addresses as numbers in our examples, I'll use a simpler notation. To indicate that a pointer variable `p` stores the address of a variable `i`, I'll show the contents of `p` as an arrow directed toward `i`:



Declaring Pointer Variables

A pointer variable is declared in much the same way as an ordinary variable. The only difference is that the name of a pointer variable must be preceded by an asterisk:

```
int *p;
```

This declaration states that p is a pointer variable capable of pointing to ***objects*** of type int. I'm using the term *object* instead of *variable* since—as we'll see in Chapter 17—p might point to an area of memory that doesn't belong to a variable. (Be aware that "object" will have a different meaning when we discuss program design in Chapter 19.)

abstract objects ➤19.1

Pointer variables can appear in declarations along with other variables:

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

In this example, i and j are ordinary integer variables, a and b are arrays of integers, and p and q are pointers to integer objects.

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. In fact, a pointer variable can even point to another pointer.

pointers to pointers ➤17.6

11.2 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. If x is a variable, then &x is the address of x in memory. To gain access to the object that a pointer points to, we use the * (***indirection***) operator. If p is a pointer, then *p represents the object to which p currently points.

The Address Operator

Declaring a pointer variable sets aside space for a pointer but doesn't make it point to an object:

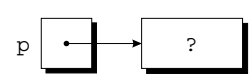
```
int *p;   /* points nowhere in particular */
```

It's crucial to initialize p before we use it. One way to initialize a pointer variable is to assign it the address of some variable—or, more generally, lvalue—using the & operator:

lvalues ➤4.2

```
int i, *p;
…
p = &i;
```

By assigning the address of i to the variable p, this statement makes p point to i:



It's also possible to initialize a pointer variable at the time we declare it:

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

Q&A

We can even combine the declaration of `i` with the declaration of `p`, provided that `i` is declared first:

```
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`, for example, we can print the value of `i` as follows:

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

Q&A

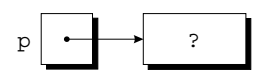
`printf` will display the *value* of `i`, not the *address* of `i`.

The mathematically inclined reader may wish to think of `*` as the inverse of `&`. Applying `&` to a variable produces a pointer to the variable; applying `*` to the pointer takes us back to the original variable:

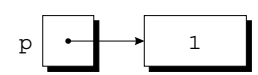
```
j = *&i;   /* same as j = i; */
```

As long as `p` points to `i`, `*p` is an ***alias*** for `i`. Not only does `*p` have the same value as `i`, but changing the value of `*p` also changes the value of `i`. (`*p` is an lvalue, so assignment to it is legal.) The following example illustrates the equivalence of `*p` and `i`; diagrams show the values of `p` and `i` at various points in the computation.

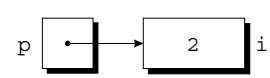
```
p = &i;
```




```
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 */
```

Never apply the indirection operator to an uninitialized pointer variable. If a pointer variable `p` hasn't been initialized, attempting to use the value of `p` in any way causes undefined behavior. In the following example, the call of `printf` may print garbage, cause the program to crash, or have some other effect:

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

Assigning a value to `*p` is particularly dangerous. If `p` happens to contain a valid memory address, the following assignment will attempt to modify the data stored at that address:

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

If the location modified by this assignment belongs to the program, it may behave erratically; if it belongs to the operating system, the program will most likely crash. Your compiler may issue a warning that `p` is uninitialized, so pay close attention to any warning messages you get.

11.3 Pointer Assignment

C allows the use of the assignment operator to copy pointers, provided that they have the same type. Suppose that `i`, `j`, `p`, and `q` have been declared as follows:

```
int i, j, *p, *q;
```

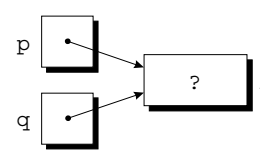
The statement

```
p = &i;
```

is an example of pointer assignment; the address of `i` is copied into `p`. Here's another example of pointer assignment:

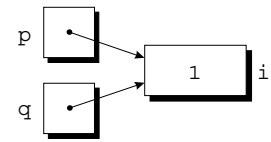
```
q = p;
```

This statement copies the contents of `p` (the address of `i`) into `q`, in effect making `q` point to the same place as `p`:

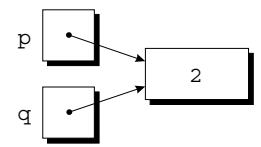


Both `p` and `q` now point to `i`, so we can change `i` by assigning a new value to either `*p` or `*q`:

```
*p = 1;
```




```
*q = 2;
```



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

Be careful not to confuse

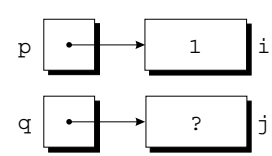
```
q = p;
```

with

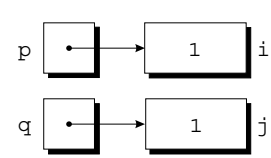
```
*q = *p;
```

The first statement is a pointer assignment; the second isn't, as the following example shows:

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




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



The assignment `*q = *p` copies the value that `p` points to (the value of `i`) into the object that `q` points to (the variable `j`).

11.4 Pointers as Arguments

So far, we've managed to avoid a rather important question: What are pointers good for? There's no single answer to that question, since pointers have several distinct uses in C. In this section, we'll see how a pointer to a variable can be useful as a function argument. We'll discover other uses for pointers in Section 11.5 and in Chapters 12 and 17.

We saw in Section 9.3 that a variable supplied as an argument in a function call is protected against change, because C passes arguments by value. This property of C can be a nuisance if we want the function to be able to modify the variable. In Section 9.3, we tried—and failed—to write a `decompose` function that could modify two of its arguments.

Pointers offer a solution to this problem: instead of passing a variable `x` as the argument to a function, we'll supply `&x`, a pointer to `x`. We'll declare the corresponding parameter `p` to be a pointer. When the function is called, `p` will have the value `&x`, hence `*p` (the object that `p` points to) will be an alias for `x`. Each appearance of `*p` in the body of the function will be an indirect reference to `x`, allowing the function both to read `x` and to modify it.

To see this technique in action, let's modify the `decompose` function by declaring the parameters `int_part` and `frac_part` to be pointers. The definition of `decompose` will now look like this:

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

The prototype for `decompose` could be either

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

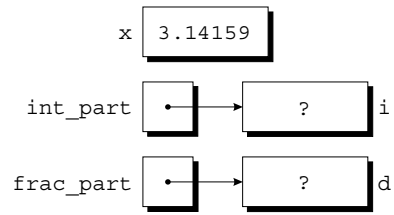
or

```
void decompose(double, long *, double *);
```

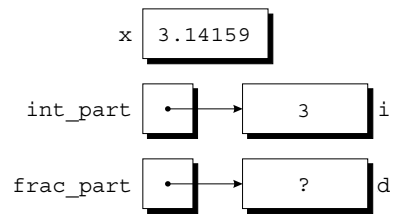
We'll call `decompose` in the following way:

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

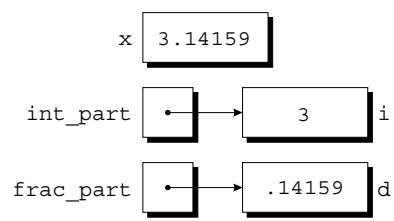
Because of the `&` operator in front of `i` and `d`, the arguments to `decompose` are *pointers* to `i` and `d`, not the *values* of `i` and `d`. When `decompose` is called, the value 3.14159 is copied into `x`, a pointer to `i` is stored in `int_part`, and a pointer to `d` is stored in `frac_part`:



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`. Since `int_part` points to `i`, the assignment puts the value 3 in `i`:



The second assignment fetches the value that `int_part` points to (the value of `i`), which is 3. This value is converted to type `double` and subtracted from `x`, giving .14159, which is then stored in the object that `frac_part` points to:



When `decompose` returns, `i` and `d` will have the values 3 and .14159, just as we originally wanted.

Using pointers as arguments to functions is actually nothing new; we've been doing it in calls of `scanf` since Chapter 2. Consider the following example:

```
int i;
…
scanf("%d", &i);
```

We must put the `&` operator in front of `i` so that `scanf` is given a *pointer* to `i`; that pointer tells `scanf` where to put the value that it reads. Without the `&`, `scanf` would be supplied with the *value* of `i`.

Although `scanf`'s arguments must be pointers, it's not always true that every argument needs the `&` operator. In the following example, `scanf` is passed a pointer variable:

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

Since `p` contains the address of `i`, `scanf` will read an integer and store it in `i`. Using the `&` operator in the call would be wrong:

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

`scanf` would read an integer and store it in `p` instead of in `i`.

Failing to pass a pointer to a function when one is expected can have disastrous results. Suppose that we call `decompose` without the `&` operator in front of `i` and `d`:

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

`decompose` is expecting pointers as its second and third arguments, but it's been given the *values* of `i` and `d` instead. `decompose` has no way to tell the difference, so it will use the values of `i` and `d` as though they were pointers. 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` (as we should always do, of course), the compiler will let us know that we're attempting to pass arguments of the wrong type. In the case of `scanf`, however, failing to pass pointers often goes undetected by the compiler, making `scanf` an especially error-prone function.

PROGRAM

Finding the Largest and Smallest Elements in an Array

To illustrate how pointers are passed to functions, let's look at a function named `max_min` that finds the largest and smallest elements in an array. When we call `max_min`, we'll pass it pointers to two variables; `max_min` will then store its answers in these variables. `max_min` has the following prototype:

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

A call of `max_min` might have the following appearance:

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

`b` is an array of integers; `N` is the number of elements in `b`. `big` and `small` are ordinary integer variables. When `max_min` finds the largest element in `b`, it stores the value in `big` by assigning it to `*max`. (Since `max` points to `big`, an assignment to `*max` will modify the value of `big`.) `max_min` stores the smallest element of `b` in `small` by assigning it to `*min`.

To test `max_min`, we'll write a program that reads 10 numbers into an array, passes the array to `max_min`, and prints the results:

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

Here's the complete program:

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]);

  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 we call a function and pass it a pointer to a variable, we normally assume that the function will modify the variable (otherwise, why would the function require a pointer?). For example, if we see a statement like

```
f(&x);
```

in a program, we'd probably expect `f` to change the value of `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. (Section 12.3 covers this point in more detail.)

We can use the word `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:

Q&A

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

This use of `const` indicates that `p` is a pointer to a "constant integer." Attempting to modify `*p` is an error that the compiler will detect.

11.5 Pointers as Return Values

We can not only pass pointers to functions but also write functions that *return* pointers. Such functions are relatively common; we'll encounter several in Chapter 13.

The following function, when given pointers to two integers, returns a pointer to whichever integer is larger:

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

When we call `max`, we'll pass pointers to two `int` variables and store the result in a pointer variable:

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

During the call of `max`, `*a` is an alias for `i`, while `*b` is an alias for `j`. If `i` has a larger value than `j`, `max` returns the address of `i`; otherwise, it returns the address of `j`. After the call, `p` points to either `i` or `j`.

Although the `max` function 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 local variable that's been declared `static`.

Never return a pointer to an *automatic* local variable:

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

The variable `i` doesn't exist once `f` returns, so the pointer to it will be invalid. Some compilers issue a warning such as "*function returns address of local variable*" in this situation.

Pointers can point to array elements, not just ordinary variables. If `a` is an array, then `&a[i]` is a pointer to element `i` of `a`. When a function has an array argument, it's sometimes useful for the function to return a pointer to one of the elements in the array. For example, the following function returns a pointer to the middle element of the array `a`, assuming that `a` has `n` elements:

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

Chapter 12 explores the relationship between pointers and arrays in considerable detail.

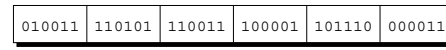
Q & A

***Q: Is a pointer always the same as an address? [p. 242]**

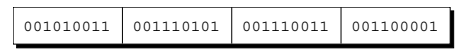
A: Usually, but not always. Consider a computer whose main memory is divided into ***words*** rather than bytes. A word might contain 36 bits, 60 bits, or some other number of bits. If we assume 36-bit words, memory will have the following appearance:

Address	Contents
0	001010011001010011001010011001010011
1	001110101001110101001110101001110101
2	001110011001110011001110011001110011
3	001100001001100001001100001001100001
4	001101110001101110001101110001101110
	⋮
n-1	001000011001000011001000011001000011

When memory is divided into words, each word has an address. An integer usually occupies one word, so a pointer to an integer can just be an address. However, a word can store more than one character. For example, a 36-bit word might store six 6-bit characters:



or four 9-bit characters:



For this reason, a pointer to a character may need to be stored in a different form than other pointers. A pointer to a character might consist of an address (the word in which the character is stored) plus a small integer (the position of the character within the word).

On some computers, pointers may be "offsets" rather than complete addresses. For example, CPUs in the Intel x86 family (used in many personal computers) can execute programs in several modes. The oldest of these, which dates back to the 8086 processor of 1978, is called ***real mode***. In this mode, addresses are sometimes represented by a single 16-bit number (an ***offset***) and sometimes by two 16-bit numbers (a ***segment:offset pair***). An offset isn't a true memory address; the CPU must combine it with a segment value stored in a special register. To support real mode, older C compilers often provide two kinds of pointers: ***near pointers*** (16-bit offsets) and ***far pointers*** (32-bit segment:offset pairs). These compilers usually reserve the words `near` and `far` as nonstandard keywords that can be used to declare pointer variables.

Q: If a pointer can point to *data* in a program, is it possible to have a pointer to program *code?

A: Yes. We'll cover pointers to functions in Section 17.7.

Q: It seems to me that there's an inconsistency between the declaration

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

and the statement

```
p = &i;
```

Why isn't `p` preceded by a `*` symbol in the statement, as it is in the declaration? [p. 244]

A: The source of the confusion is the fact that the `*` symbol can have different meanings in C, depending on the context in which it's used. In the declaration

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

the `*` symbol is *not* the indirection operator. Instead, it helps specify the type of `p`, informing the compiler that `p` is a *pointer* to an `int`. When it appears in a statement,

however, the * symbol performs indirection (when used as a unary operator). The statement

```
*p = &i;   /*** WRONG ***/
```

would be wrong, because it assigns the address of i to the object that p points to, not to p itself.

Q: Is there some way to print the address of a variable? [p. 244]

A: Any pointer, including the address of a variable, can be displayed by calling the printf function and using %p as the conversion specification. See Section 22.3 for details.

Q: The following declaration is confusing:

```
void f(const int *p);
```

Does this say that f can't modify p? [p. 251]

A: No. It says that f can't change the integer that p *points to*; it doesn't prevent f from changing p itself.

```
void f(const int *p)
{
  int j;

  *p = 0;   /*** WRONG ***/
  p = &j;   /* legal */
}
```

Since arguments are passed by value, assigning p a new value—by making it point somewhere else—won't have any effect outside the function.

***Q: When declaring a parameter of a pointer type, is it legal to put the word const in front of the parameter's name, as in the following example?**

```
void f(int * const p);
```

A: Yes, although the effect isn't the same as if const precedes p's type. We saw in Section 11.4 that putting const *before* p's type protects the object that p points to. Putting const *after* p's type protects p itself:

```
void f(int * const p)
{
  int j;

  *p = 0;   /* legal */
  p = &j;   /*** WRONG ***/
}
```

This feature isn't used very often. Since p is merely a copy of another pointer (the argument when the function is called), there's rarely any reason to protect it.

An even greater rarity is the need to protect both p *and* the object it points to, which can be done by putting const both before and after p's type:

```
void f(const int * const p)
{
  int j;

  *p = 0;   /*** WRONG ***/
  p = &j;   /*** WRONG ***/
}
```

Exercises

Section 11.2

1. If `i` is a variable and `p` points to `i`, which of the following expressions are aliases for `i`?

   (a) `*p` (c) `*&p` (e) `*i` (g) `*&i`
   (b) `&p` (d) `&*p` (f) `&i` (h) `&*i`

Section 11.3

W 2. If `i` is an `int` variable and `p` and `q` are pointers to `int`, which of the following assignments are legal?

   (a) `p = i;` (d) `p = &q;` (g) `p = *q;`
   (b) `*p = &i;` (e) `p = *&q;` (h) `*p = q;`
   (c) `&p = q;` (f) `p = q;` (i) `*p = *q;`

Section 11.4

3. The following function supposedly computes the sum and average of the numbers in the array `a`, which has length `n`. `avg` and `sum` point to variables that the function should modify. Unfortunately, the function contains several errors; find and correct them.

```
void avg_sum(double a[], int n, double *avg, double *sum)
{
  int i;

  sum = 0.0;
  for (i = 0; i < n; i++)
    sum += a[i];
  avg = sum / n;
}
```

W 4. Write the following function:

```
void swap(int *p, int *q);
```

When passed the addresses of two variables, `swap` should exchange the values of the variables:

```
swap(&i, &j);   /* exchanges values of i and j */
```

5. Write the following function:

```
void split_time(long total_sec, int *hr, int *min, int *sec);
```

`total_sec` is a time represented as the number of seconds since midnight. `hr`, `min`, and `sec` are pointers to variables in which the function will store the equivalent time in hours (0–23), minutes (0–59), and seconds (0–59), respectively.

W 6. Write the following function:

```
void find_two_largest(int a[], int n, int *largest,
                      int *second_largest);
```

When passed an array `a` of length `n`, the function will search `a` for its largest and second-largest elements, storing them in the variables pointed to by `largest` and `second_largest`, respectively.

7. Write the following function:

```
void split_date(int day_of_year, int year,
                int *month, int *day);
```

`day_of_year` is an integer between 1 and 366, specifying a particular day within the year designated by `year`. `month` and `day` point to variables in which the function will store the equivalent month (1–12) and day within that month (1–31).

Section 11.5

8. Write the following function:

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

When passed an array `a` of length `n`, the function will return a pointer to the array's largest element.

Programming Projects

1. Modify Programming Project 7 from Chapter 2 so that it includes the following function:

```
void pay_amount(int dollars, int *twenties, int *tens,
                int *fives, int *ones);
```

The function determines the smallest number of $20, $10, $5, and $1 bills necessary to pay the amount represented by the `dollars` parameter. The `twenties` parameter points to a variable in which the function will store the number of $20 bills required. The `tens`, `fives`, and `ones` parameters are similar.

2. Modify Programming Project 8 from Chapter 5 so that it includes the following function:

```
void find_closest_flight(int desired_time,
                         int *departure_time,
                         int *arrival_time);
```

This function will find the flight whose departure time is closest to `desired_time` (expressed in minutes since midnight). It will store the departure and arrival times of this flight (also expressed in minutes since midnight) in the variables pointed to by `departure_time` and `arrival_time`, respectively.

3. Modify Programming Project 3 from Chapter 6 so that it includes the following function:

```
void reduce(int numerator, int denominator,
            int *reduced_numerator,
            int *reduced_denominator);
```

`numerator` and `denominator` are the numerator and denominator of a fraction. `reduced_numerator` and `reduced_denominator` are pointers to variables in which the function will store the numerator and denominator of the fraction once it has been reduced to lowest terms.

4. Modify the `poker.c` program of Section 10.5 by moving all external variables into `main` and modifying functions so that they communicate by passing arguments. The `analyze_hand` function needs to change the `straight`, `flush`, `four`, `three`, and `pairs` variables, so it will have to be passed pointers to those variables.