POINTERS

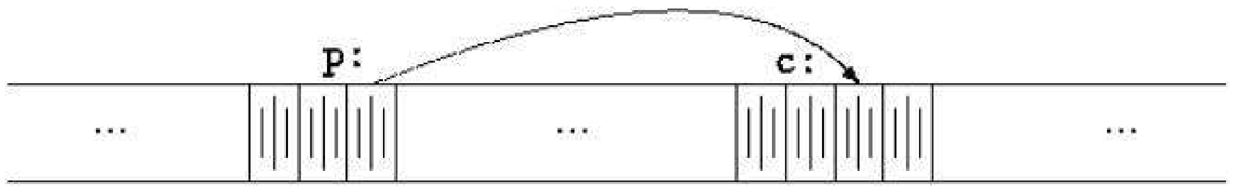
A pointer is a variable that contains the address of a variable. Pointers are much used in C, partly because they are sometimes the only way to express a computation, and partly because they usually lead to more compact and efficient code than can be obtained in other ways. Pointers and arrays are closely related; this chapter also explores this relationship and shows how to exploit it.

Pointers have been lumped with the `goto` statement as a marvelous way to create impossible to understand programs. This is certainly true when they are used carelessly, and it is easy to create pointers that point somewhere unexpected. With discipline, however, pointers can alsobe used to achieve clarity and simplicity. This is the aspect that we will try to illustrate.

The main change in ANSI C is to make explicit the rules about how pointers can be manipulated, in effect mandating what good programmers already practice and good compilers already enforce. In addition, the type `void *` (pointer to `void`) replaces `char *` as the proper type for a generic pointer.

Pointers and Addresses

Let us begin with a simplified picture of how memory is organized. A typical machine has an array of consecutively numbered or addressed memory cells that may be manipulated individually or in contiguous groups. One common situation is that any byte can be a `char`, a pair of one-byte cells can be treated as a `short` integer, and four adjacent bytes form a `long`. A pointer is a group of cells (often two or four) that can hold an address. So if `c` is a `char` and `p` is a pointer that points to it, we could represent the situation this way:



The unary operator `&`gives the address of an object, so the statement

```
p = &c;
```

assigns the address of `c` to the variable `p`, and `p` is said to ``point to" `c`. The `&`operator only applies to objects in memory: variables and array elements. It cannot be applied to expressions, constants, or `register` variables.

The unary operator `*` is the *indirection* or *dereferencing* operator; when applied to a pointer, it accesses the object the pointer points to. Suppose that `x` and `y` are integers and `ip`is a pointer to `int`. This artificial sequence shows how to declare a pointer and how to use `&`and `*`:

```
int x = 1, y = 2, z[10];
int *ip;
ip = &x;
y = *ip;
*ip = 0;
ip = &z[0];
```

The declaration of `x`, `y`, and `z` are what we've seen all along. The declaration of the pointer `ip`.

```
int *ip;
```

is intended as a mnemonic; it says that the expression `*ip`is an `int`. The syntax of the declaration for a variable mimics the syntax of expressions in which the variable might appear. This reasoning applies to function declarations as well. For example,

```
double *dp, atof(char *);
```

says that in an expression `*dp`and `atof(s)` have values of `double`, and that the argument of `atof`is a pointer to `char`.
You should also note the implication that a pointer is constrained to point to a particular kind of object: every pointer points to a specific data type. If `ip`points to the integer `x`, then `*ip`can occur in any context where `x` could, so

```
*ip = *ip + 10;
```

increments `*ip` by 10.
The unary operators `*` and `&`bind more tightly than arithmetic operators, so the assignment

```
y = *ip + 1
```

takes whatever `ip`points at, adds 1, and assigns the result to `y`, while

```
*ip += 1
```

increments what `ip`points to, as do

`++*ip` and `(*ip)++`

The parentheses are necessary in this last example; without them, the expression would increment `ip` instead of what it points to, because unary operators like `*` and `++` associate right to left.
Finally, since pointers are variables, they can be used without dereferencing. For example, if `iq` is another pointer to `int`,

```
iq = ip
```

copies the contents of `ip`into `iq`, thus making `iq`point to whatever `ip`pointed to.

Pointers and Function Arguments

Since C passes arguments to functions by value, there is no direct way for the called function to alter a variable in the calling function. For instance, a sorting routine might exchange two out-oforder arguments with a function called `swap`. It is not enough to write

```
swap(a, b);
```

where the `swap` function is defined as

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

Because of call by value, `swap` can't affect the arguments `a` and `b` in the routine that called it. The function above swaps *copies* of `a` and `b`. The way to obtain the desired effect is for the calling program to pass pointers to the values to be changed:

```
swap(&a, &b);
```

Since the operator & produces the address of a variable, &a is a pointer to a. In swap itself, the parameters are declared as pointers, and the operands are accessed indirectly through them.

```
void swap(int *px, int *py) /* interchange *px and *py */

{

 int temp;

 temp = *px;

 *px = *py;

 *py = temp;

}
```

Pointer arguments enable a function to access and change objects in the function that called it. As an example, consider a function `getint` that performs free-format input conversion by breaking a stream of characters into integer values, one integer per call. `getint` has to return the value it found and also signal end of file when there is no more input. These values have to be passed back by separate paths, for no matter what value is used for `EOF`, that could also be the value of an input integer.

One solution is to have `getint` return the end of file status as its function value, while using a pointer argument to store the converted integer back in the calling function. This is the scheme used by `scanf`as well

The following loop fills an array with integers by calls to `getint`:

```
int n, array[SIZE], getint(int *);
for (n = 0; n < SIZE &&getint(&array[n]) != EOF; n++)
;
```

Each call sets `array[n]` to the next integer found in the input and increments `n`. Notice that it is essential to pass the address of `array[n]` to `getint`. Otherwise there is no way for `getint` to communicate the converted integer back to the caller.

Our version of `getint`returns `EOF` for end of file, zero if the next input is not a number, and a positive value if the input contains a valid number.

```
#include <ctype.h>
int getch(void);
void ungetch(int);
int getint(int *pn)
{
 int c, sign;
 while (isspace(c = getch()));
 if (!isdigit(c) && c != EOF && c != '+' && c != '-')
 {
  ungetch(c); return 0;
 }
 sign = (c == '-') ? -1 : 1;
 if (c == '+' || c == '-')
  c = getch();
for (*pn = 0; isdigit(c), c = getch())
*pn = 10 * *pn + (c - '0');
*pn *= sign;
 if (c != EOF)
 ungetch(c);
 return c;
}
```

Throughout `getint`, `*pn`is used as an ordinary `int`variable. We have also used `getch`and `ungetch`so the one extra character that must be read can be pushed back onto the input.