5.3. Pointers

Using pointers is a bit like riding a bicycle. Just when you think that you'll never understand them—suddenly you do! Once learned the trick is hard to forget. There's no real magic to pointers, and a lot of readers will already be familiar with their use. The only peculiarity of C is how heavily it relies on the use of pointers, compared with other languages, and the relatively permissive view of what you can do with them.

5.3.1. Declaring pointers

Of course, just like other variables, you have to declare pointers before you can use them. Pointer declarations look much like other declarations: but don't be misled. When pointers are declared, the keyword at the beginning (c int, char and so on) declares the type of variable that the pointer will point to. The pointer itself is not of that type, it is of type pointer to that type. A given pointer only points to one particular type, not to all possible types. Here's the declaration of an array and a pointer:

```
int ar[5], *ip;
```

We now have an array and a pointer (see Figure 5.3):

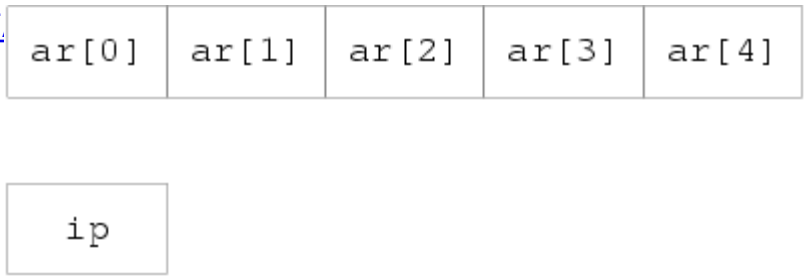

Figure 5.3. An array and a pointer

The `*` in front of `ip` in the declaration shows that it is a pointer, not an ordinary variable. It is of type `pointer to int`, and can only be used to refer to variables of type int. It's still uninitialized, so to do anything useful with it, it has to be made to point to something. You can't just stick some integer value into it, because integer values have the type `int`, not `pointer to int`, which is what we want. (In any case, what would it mean if this fragment were valid:

```
ip = 6;
```

What would `ip` be pointing to? In fact it could be construed to have a number of meanings, but the simple fact is that, in C, that sort of thing is just wrong.)

Here is the right way to initialize a pointer:

```
int ar[5], *ip;
ip = &ar[3];
```

In that example, the pointer is made to point to the member of the array `ar` whose index is `3`, i.e. the fourth member. This is important. You can assign values to pointers just like ordinary variables; the difference is simply in what the value means. The values of the variables that we have now are shown in Figure 5.4 (`??` means uninitialized).

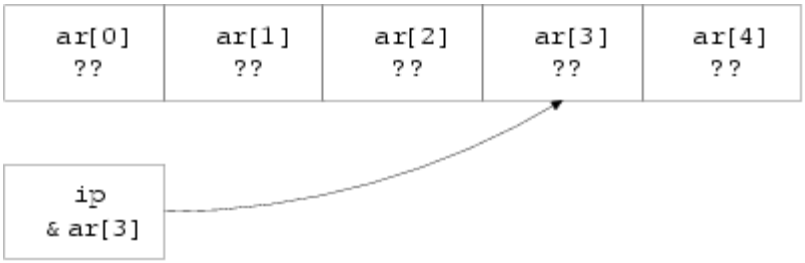

Figure 5.4. Array and initialized pointer

You can see that the variable `ip` has the value of the expression `&ar[3]`. The arrow indicates that, when used as a pointer, `ip` points to the variable `ar[3]`.

What is this new unary `&`? It is usually described as the ‘address-of’ operator, since on many systems the pointer will hold the store address of the thing that it points to. If you understand what addresses are, then you will probably have more trouble than those who don't: thinking about pointers as if they were addresses generally leads to grief. What seems a perfectly reasonable address manipulation on processor X can almost always be shown to be impossible on manufacturer Y's washing machine controller which uses 17-bit addressing when it's on the spin cycle, and reverses the order of odd and even bits when it's out of bleach. (Admittedly, it's unlikely that **anyone** could get C to work an an architecture like that. But you should see some of the ones it **does** work on; they aren't much better.)

We will continue to use the term ‘address of’ though, because to invent a different one would be even worse.

Applying the `&` operator to an operand returns a pointer to the operand:

```
int i;
float f;
      /* '&i' would be of type pointer to int */
```

```
        /* '&f' would be of type pointer to float */
```

In each case the pointer would point to the object named in the expression.

A pointer is only useful if there's some way of getting at the thing that it points to; C uses the unary `*` operator for this job. If `p` is of type ‘pointer to something’, then *p refers to the thing that is being pointed to. For example, to access the variable `x` via the pointer `p`, this would work:

```
#include <stdio.h>
#include <stdlib.h>
main(){
      int x, *p;

      p = &x;         /* initialise pointer */
      *p = 0;         /* set x to zero */
      printf("x is %d\n", x);
      printf("*p is %d\n", *p);

      *p += 1;        /* increment what p points to */
      printf("x is %d\n", x);

      (*p)++;         /* increment what p points to */
      printf("x is %d\n", x);

      exit(EXIT_SUCCESS);
}
```

Example 5.1

You might be interested to note that, since `&` takes the address of an object, returning a pointer to it, and since `*` means ‘the thing pointed to by the pointer’, the `&` and `*` in the combination `*&` effectively cancel each other out. (But be careful. Some things, constants for example, don't have addresses and the `&` operator cannot be applied to them; `&1.5` is not a pointer to anything, it's an error.) It's also interesting to see that C is one of the few languages that allows an expression on the left-hand side of an assignment operator. Look back at the

example: the expression `*p` occurs twice in that position, and then the amazing `(*p)++;` statement. That last one is a great puzzle to most beginners—even if you've managed to wrap your mind around the concept that `*p = 0` writes zero into the thing pointed to by `p`, and that `*p += 1` adds one to where `p` points, it still seems a bit much to apply the `++` operator to `*p`.

The precedence of `(*p)++` deserves some thought. It will be given more later, but for the moment let's work out what happens. The brackets ensure that the * applies to p, so what we have is 'post-increment the thing pointed to by p'. Looking at Table 2.9, it turns out that ++ and * have equal precedence, but they associate right to left; in other words, without the brackets, the implied operation would have been *(p++), whatever that would mean. Later on you'll be more used to it—for the moment, we'll be careful with brackets to show the way that those expressions work.

So, provided that a pointer holds the address of something, the notation `*pointer` is equivalent to giving the name of the something directly. What benefit do we get from all this? Well, straight away it gets round the call-by-value restriction of functions. Imagine a function that has to return, say, two integers representing a month and a day within that month. The function has some (unspecified) way of determining these values; the hard thing to do is to return two separate values. Here's a skeleton of the way that it can be done:

```
#include <stdio.h>
#include <stdlib.h>
void
```

```
date(int *, int *);     /* declare the function */

main(){
      int month, day;
      date (&day, &month);
      printf("day is %d, month is %d\n", day, month);
      exit(EXIT_SUCCESS);
}

void
date(int *day_p, int *month_p){
      int day_ret, month_ret;
      /*
       * At this point, calculate the day and month
       * values in day_ret and month_ret respectively.
       */
      *day_p = day_ret;
      *month_p = month_ret;
}
```

Example 5.2

Notice carefully the advance declaration of `date` showing that it takes two arguments of type 'pointer to `int`'. It returns `void`, because the values are passed back via the pointers, not the usual return value. The `main` function passes pointers as arguments to date, which first uses the internal variables `day_ret` and `month_ret` for its calculations, then takes those values and assigns them to the places pointed to by its arguments.

When `date` is called, the situation looks like Figure 5.5.

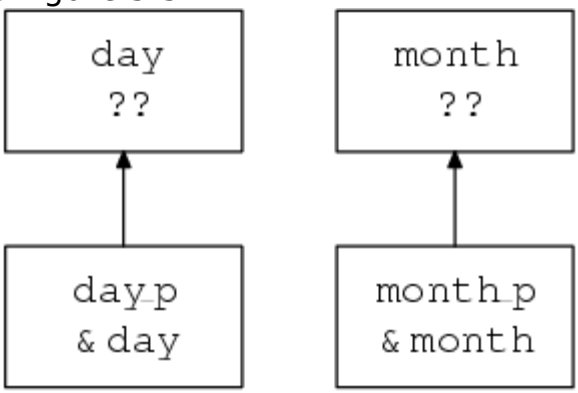

Figure 5.5. Just as `date` is called

The arguments have been passed to `date`, but in `main`, day and month are uninitialized. When date reaches the return statement, the situation is as shown in Figure 5.6 (assuming that the values for day and month are 12 and 5 respectively).

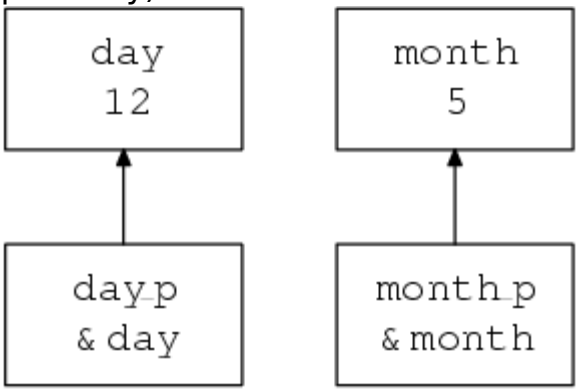

Figure 5.6. Just as `date` is about to return

One of the great benefits introduced by the new Standard is that it allows the types of the arguments to date to be declared in advance. A great favourite (and disastrous) mistake in C is to forget that a function expects pointers as its arguments, and to pass something else instead. Imagine what would have happened if the call of date above had read

```
date(day, month);
```

and no previous declaration of date had been visible. The compiler would not have known that date expects pointers as arguments, so it would pass the `int` values of `day` and `month` as the arguments. On a large number of computers, pointers and integers can be passed in the same way, so the function would execute, then pass back its return values by putting them into wherever `day` and `month` would point if their contents were pointers. This is very unlikely to give any

sensible results, and in general causes unexpected corruption of data elsewhere in the computer's store. It can be extremely hard to track down!

Fortunately, by declaring `date` in advance, the compiler has enough information to warn that a mistake has almost certainly been made.

Perhaps surprisingly, it isn't all that common to see pointers used to give this call-by-reference functionality. In the majority of cases, call-by-value and a single return value are adequate. What is **much** more common is to use pointers to 'walk' along arrays.

5.3.2. Arrays and pointers

Array elements are just like other variables: they have addresses.

```
int ar[20], *ip;

ip = &ar[5];
*ip = 0;        /* equivalent to ar[5] = 0; */
```

The address of `ar[5]` is put into `ip`, then the place pointed to has zero assigned to it. By itself, this isn't particularly exciting. What **is** interesting is the way that pointer arithmetic works. Although it's simple, it's one of the cornerstones of C.

Adding an integral value to a pointer results in another pointer of the same type. Adding **n** gives a pointer which points n elements further along an array than the original pointer did. (Since **n** can be negative, subtraction is obviously possible too.) In the example above, a statement of the form

```
*(ip+1) = 0;
```

would set `ar[6]` to zero, and so on. Again, this is not obviously any improvement on 'ordinary' ways of accessing an array, but the following is.

```
int ar[20], *ip;

for(ip = &ar[0]; ip < &ar[20]; ip++)
      *ip = 0;
```

That example is a classic fragment of C. A pointer is set to point to the start of an array, then, while it still points inside the array, array elements are accessed one by one, the pointer incrementing between each one. The Standard endorses existing practice by guaranteeing that it's permissible to use the **address** of `ar[20]` even though no such element exists. This allows you to use it for checks in loops like the one above. The guarantee only extends to one element beyond the end of an array and no further.

Why is the example better than indexing? Well, most arrays are accessed sequentially. Very few programming examples actually make use of the 'random access' feature of arrays. If you do just want sequential access, using a pointer can give a worthwhile improvement in speed. In terms of the underlying address arithmetic, on most architectures it takes one multiplication and one addition to access a one-dimensional array through a subscript. Pointers require no arithmetic at all—they nearly always hold the store address of the object that they refer to. In the example above, the only arithmetic that has to be done is in the `for` loop, where one comparison and one

addition are done each time round the loop. The equivalent, using indexes, would be this:

```
int ar[20], i;
for(i = 0; i < 20; i++)
      ar[i] = 0;
```

The same amount of arithmetic occurs in the loop statement, but an extra address calculation has to be performed for every array access.

Efficiency is not normally an important issue, but here it can be. Loops often get traversed a substantial number of times, and every microsecond saved in a big loop can matter. It isn't always easy for even a smart compiler to recognize that this is the sort of code that could be ‘pointerized’ behind the scenes, and to convert from indexing (what the programmer wrote) to actually use a pointer in the generated code.

If you have found things easy so far, read on. If not, it's a good idea to skip to Section 5.3.3. What follows, while interesting, isn't essential. It has been known to frighten even experienced C programmers.

To be honest, C doesn't really ‘understand’ array indexing, except in declarations. As far as the compiler is concerned, an expression like `x[n]` is translated into `*(x+n)` and use made of the fact that an array name is converted into a pointer to the array's first element whenever the name occurs in an expression. That's why, amongst other things, array elements count from zero: if `x` is an array name, then in an expression, `x` is equivalent to `&x[0]`, i.e. a pointer to the first element of the array. So, since

`*(&x[0])` uses the pointer to get to `x[0]`, `*(&x[0] + 5)` is the same as `*(x + 5)` which is the same as `x[5]`. A curiosity springs out of all this. If `x[5]` is translated into `*(x + 5)`, and the expression `x + 5` gives the same result as `5 + x` (it does), then `5[x]` should give the identical result to `x[5]`! If you don't believe that, here is a program that compiles and runs successfully:

```
#include <stdio.h>
#include <stdlib.h>
#define ARSZ 20
main(){
      int ar[ARSZ], i;
      for(i = 0; i < ARSZ; i++){
              ar[i] = i;
              i[ar]++;
              printf("ar[%d] now = %d\n", i, ar[i]);
      }

      printf("15[ar] = %d\n", 15[ar]);
      exit(EXIT_SUCCESS);
}
```

Example 5.3

Summary

- Arrays always index from zero—end of story.
- There are no multidimensional arrays; you use arrays of arrays instead.
- Pointers point to things; pointers to different types are themselves different types. They have nothing in common with each other or any other types in C; there are no automatic conversions between pointers and other types.
- Pointers can be used to simulate ‘call by reference’ to functions, but it takes a little work to do it.
- Incrementing or adding something to a pointer can be used to step along arrays.

- To facilitate array access by incrementing pointers, the Standard guarantees that in an *n* element array, although element *n* does not exist, use of its address is not an error—the valid range of addresses for an array declared as `int ar[N]` is `&ar[0]` through to `&ar[N]`. You must not try to access this last pseudo-element.

5.3.3. Qualified types

If you are confident that you have got a good grasp of the basic declaration and use of pointers we can continue. If not, it's important to go back over the previous material and make sure that there is nothing in it that you still find obscure; although what comes next looks more complicated than it really is, there's no need to make it worse by starting unprepared.

The Standard introduces two things called *type qualifiers*, neither of which were in Old C. They can be applied to any declared type to modify its behaviour —hence the term ‘qualifier’—and although one of them can be ignored for the moment (the one named `volatile`), the other, `const`, cannot.

If a declaration is prefixed with the keyword `const`, then the thing that is declared is announced to the world as being constant. You must not attempt to modify (change the value of) `const` objects, or you get undefined behaviour. Unless you have used some very dirty tricks, the compiler will know that the thing you are trying to modify is constant, so it can warn you.

There are two benefits in being able to declare things to be `const`.

1. It documents the fact that the thing is unmodifiable and the compiler helps to check. This is especially reassuring in the case of functions which take pointers as arguments. If the declaration of a function shows that the arguments are pointers to constant objects, then you know that the function is not allowed to change them through the pointers.
2. If the compiler knows that things are constant, it can often do increased amounts of optimization or generate better code.

Of course, constants are not much use unless you can assign an initial value to them. We won't go into the rules about initialization here (they are in Chapter 6), but for the moment just note that any declaration can also assign the value of a constant expression to the thing being declared. Here are some example declarations involving const:

```
const int x = 1;        /* x is constant */
const float f = 3.5;    /* f is constant */
const char y[10];       /* y is an array of 10 const ints */
                        /* don't think about initializing it yet! */
```

What is more interesting is that pointers can have this qualifier applied in two ways: either to the thing that it points to (pointer to const), or to the pointer itself (constant pointer). Here are examples of **that**:

```
int i;                  /* i is an ordinary int */
const int ci = 1;       /* ci is a constant int */
int *pi;                /* pi is a pointer to an int */
const int *pci;         /* pc is a pointer to a constant int */
      /* and now the more complicated stuff */
```

```
/* cpi is a constant pointer to an int */
int *const cpi = &i;

/* cpci is a constant pointer to an constant int */
const int *const cpci = &ci;
```

The first declaration (of `i`) is unsurprising. Next, the declaration of `ci` shows that it is a constant integer, and therefore may not be modified. If we didn't initialize it, it would be pretty well useless.

It isn't hard to understand what a pointer to an integer and a pointer to a constant integer do—but note that they are different types of pointer now and can't be freely intermixed. You can change the values of both `pi` and `pci` (so that they point to other things); you can change the value of the thing that `pi` points to (it's not a constant integer), but you are only allowed to inspect the value of the thing that `pci` points to because that is a constant.

The last two declarations are the most complicated. If the pointers themselves are constant, then you are not allowed to make them point somewhere else—so they need to be initialized, just like `ci`. Independent of the `const` or other status of the pointer itself, naturally the thing that it points to can also be `const` or non-`const`, with the appropriate constraints on what you can do with it.

A final piece of clarification: what constitutes a qualified type? In the example, `ci` was clearly of a qualified type, but pci was not, since the pointer was not qualified, only the thing that it points to. The only things that had qualified type in that list were: `ci`, `cpi`, and `cpci`.

Although the declarations do take some

mental gymnastics to understand, it just takes a little time to get used to seeing them, after which you will find that they seem quite natural. The complications come later when we have to explain whether or not you are allowed to (say) compare an ordinary pointer with a constant pointer, and if so, what does it mean? Most of those rules are 'obvious' but they do have to be stated.

Type qualifiers are given a further airing in Chapter 8.

5.3.4. Pointer arithmetic

Although a more rigorous description of pointer arithmetic is given later, we'll start with an approximate version that will do for the moment.

Not only can you add an integral value to a pointer, but you can also compare or subtract two pointers of the same type. They must both point into the same array, or the result is undefined. The difference between two pointers is defined to be the number of array elements separating them; the type of this difference is implementation defined and will be one of `short`, `int`, or `long`. This next example shows how the difference can be calculated and used, but before you read it, you need to know an important point.

In an expression the name of an array is converted to a pointer to the first element of the array. The only places where that is not true are when an array name is used in conjunction with `sizeof`, when a string is used to initialize an array or when the array name is the subject of the

address-of operator (unary `&`). We haven't seen any of those cases yet, they will be discussed later. Here's the example.

```
#include <stdio.h>
#include <stdlib.h>
#define ARSZ 10

main(){
      float fa[ARSZ], *fp1, *fp2;

      fp1 = fp2 = fa; /* address of first element */
      while(fp2 != &fa[ARSZ]){
              printf("Difference: %d\n", (int)(fp2-fp1));
              fp2++;
      }
      exit(EXIT_SUCCESS);
}
```

Example 5.4

The pointer `fp2` is stepped along the array, and the difference between its current and original values is printed. To make sure that `printf` isn't handed the wrong type of argument, the difference between the two pointers is forced to be of type `int` by using the cast `(int)`. That allows for machines where the difference between two pointers is specified to be long.

Unfortunately, if the difference does happen to be `long` and the array is enormous, the last example may give the wrong answers. This is a safe version, using a cast to force a long value to be passed:

```
#include <stdio.h>
#define ARSZ 10

main(){
      float fa[ARSZ], *fp1, *fp2;

      fp1 = fp2 = fa; /* address of first element */
      while(fp2 != &fa[ARSZ]){
              printf("Difference: %ld\n", (long)(fp2-fp1));
              fp2++;
      }
```

```
        return(0);
}
```

Example 5.5

5.3.5. void, null and dubious pointers

C is careful to keep track of the type of each pointer and will not in general allow you to use pointers of different types in the same expression. A pointer to char is a different type of pointer from a pointer to `int` (say) and you cannot assign one to the other, compare them, substitute one for the other as an argument to a function .... in fact they may even be stored differently in memory and even be of different lengths.

Pointers of different types are not the same. There are no implicit conversions from one to the other (unlike the arithmetic types).

There are a few occasions when you **do** want to be able to sidestep some of those restrictions, so what can you do?

The solution is to use the special type, introduced for this purpose, of 'pointer to `void`'. This is one of the Standard's invented features: before, it was tacitly assumed that 'pointer to `char`' was adequate for the task. This has been a reasonably successful assumption, but was a rather untidy thing to do; the new solution is both safer and less misleading. There isn't any other use for a pointer of that type—`void *` can't actually point to anything—so it improves readability. A pointer of type `void *` can have the value of any other pointer assigned to and can, conversely, be assigned to any other

pointer. This must be used with great care, because you can end up in some heinous situations. We'll see it being used safely later with the malloc library function.

You may also on occasion want a pointer that is guaranteed not to point to any object—the so-called *null pointer*. It's common practice in C to write routines that return pointers. If, for some reason, they can't return a valid pointer (perhaps in case of an error), then they will indicate failure by returning a null pointer instead. An example could be a table lookup routine, which returns a pointer to the object searched for if it is in the table, or a null pointer if it is not.

How do you write a null pointer? There are two ways of doing it and both of them are equivalent: either an integral constant with the value of `0` or that value converted to type `void *` by using a cast. Both versions are called the *null pointer constant*. If you assign a null pointer constant to any other pointer, or compare it for equality with any other pointer, then it is first converted the type of that other pointer (neatly solving any problems about type compatibility) and will not appear to have a value that is equal to a pointer to any object in the program.

The only values that can be assigned to pointers apart from 0 are the values of other pointers of the same type. However, one of the things that makes C a useful replacement for assembly language is that it allows you to do the sort of things that most other languages prevent. Try this:

```
int *ip;
```

```
ip = (int *)6;
*ip = 0xFF;
```

What does that do? The pointer has been initialized to the value of 6 (notice the cast to turn an integer 6 into a pointer). This is a highly machine-specific operation, and the bit pattern that ends up in the pointer is quite possibly nothing like the machine representation of 6. After the initialization, hexadecimal `FF` is written into wherever the pointer is pointing. The int at location 6 has had `0xFF` written into it—subject to whatever 'location 6' means on this particular machine.

It may or may not make sense to do that sort of thing; C gives you the power to express it, it's up to you to get it right. As always, it's possible to do things like this by accident, too, and to be **very** surprised by the results.

Previous section | Chapter contents | Next section