Subsections

Advanced Pointer Topics

We have introduced many applications and techniques that use pointers. We have introduced some advanced pointer issues already. This chapter brings together some topics we have briefly mentioned and others to complete our study C pointers.

In this chapter we will:

- Examine pointers to pointers in more detail.
- See how pointers are used in command line input in C.
- Study pointers to functions

Pointers to Pointers

We introduced the concept of a pointer to a pointer previously. You can have a pointer to a pointer of any type.

Consider the following:

```
char ch;  /* a character */
char *pch; /* a pointer to a character */
char **ppch; /* a pointer to a pointer to a character */
```

We can visualise this in Figure 11.1. Here we can see that `**ppch` refers to memory address of `*pch` which refers to the memory address of the variable `ch`. But what does this mean in practice?

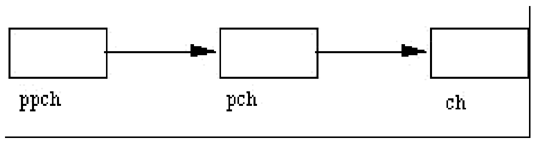

Fig. 11.1 Pointers to pointers Recall that `char *` refers to a (`NULL` terminated string. So one common and convenient notion is to declare a pointer to a pointer to a string (Figure 11.2)

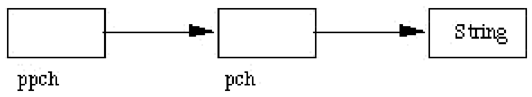

Fig. 11.2 Pointer to String Taking this one stage further we can have several strings being pointed to by the pointer (Figure 11.3)

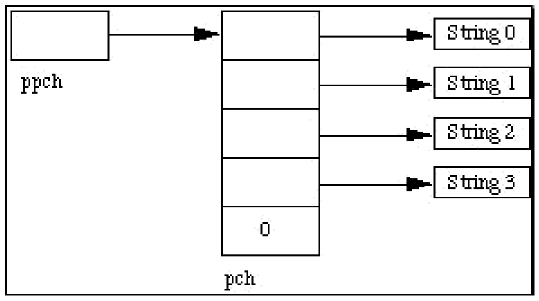

Fig. 11.3 Pointer to Several Strings We can refer to individual strings by `ppch[0], ppch[1],` . . . . . Thus this is identical to declaring `char *ppch[]`.

One common occurrence of this type is in C command line argument input which we now consider.

Command line input

C lets read arguments from the command line which can then be used in our programs.

We can type arguments after the program name when we run the program.

We have seen this with the compiler for example

```
 c89 -o prog prog.c
```

`c89` is the program, `-o prog prog.c` the arguments.

In order to be able to use such arguments in our code we must define them as follows:

```
 main(int argc, char **argv)
```

So our `main` function now has its own arguments. These are the only arguments main accepts.

- **argc** is the number of arguments typed -- including the program name.
- **argv** is an array of strings holding each command line argument -- including the program name in the first array element.

A simple program example:

```
#include<stdio.h>

main (int argc, char **argv)
    { /* program to print arguments
                                 from command line */

                                 int i;

                                 printf(``argc = %d\n\n'',argc);

                                 for (i=0;i<argc;++i)
                                                 printf(``argv[%d]: %s\n'',

                                                                 i, argv[i]);
                 }
```

Assume it is compiled to run it as args.

So if we type:

```
 args f1 ``f2'' f3 4 stop!
```

```
The output would be:


   argc = 6

                 argv[0] = args
                 argv[1] = f1
                 argv[2] = f2
                 argv[3] = f3
                 argv[4] = 4
                 argv[5] = stop!


NOTE: ● argv[0] is program name.
```

```
● argc counts program name
● Embedded `` '' are ignored.
Blank spaces delimit end of arguments.
Put blanks in `` '' if needed.
```

Pointers to a Function

Pointer to a function are perhaps on of the more confusing uses of pointers in C. Pointers to functions are not as common as other pointer uses. However, one common use is in a passing pointers to a function as a parameter in a function call. (Yes this is getting confusing, hold on to your hats for a moment).

This is especially useful when alternative functions maybe used to perform similar tasks on data. You can pass the data and the function to be used to some ***control*** function for instance. As we will see shortly the C standard library provided some basic sorting (`qsort`) and searching (`bsearch`) functions for free. You can easily embed your own functions.

To declare a pointer to a function do:

```
int (*pf) ();
```

This simply declares a pointer `*pf` to function that returns and `int`. No actual function is ***pointed*** to yet.

If we have a function `int f()` then we may simply (!!) write:

```
pf = &f;
```

For compiler prototyping to fully work it is better to have full function prototypes for the function and the pointer to a function:

```
int f(int);
int (*pf) (int) = &f;
```

Now `f()` returns an `int` and takes one `int` as a parameter.

You can do things like:

```
ans = f(5);
ans = pf(5);
```

which are equivalent.

The `qsort` standard library function is very useful function that is designed to sort an array by a ***key*** value of ***any type*** into ascending order, as long as the elements of the array are of fixed type.

qsort is prototyped in (`stdlib.h`):

```
void qsort(void *base, size_t num_elements, size_t element_size,
   int (*compare)(void const *, void  const *));
```

The argument `base` points to the array to be sorted, `num_elements` indicates how long the array is, `element_size` is the size in bytes of each array element and the final argument `compare` is a pointer to a function.

`qsort` calls the `compare` function which is user defined to compare the data when sorting. Note that `qsort` maintains it's data type independence by giving the comparison responsibility to the user. The compare function must return certain (`integer`) values according to the comparison result:

less than zero
: if first value is less than the second value

zero

: if first value is equal to the second value

greater than zero

: if first value is greater than the second value

Some quite complicated data structures can be sorted in this manner. For example, to sort the following structure by `integer` key:

```
typedef struct {
        int   key;
                                                                struct other_data;
} Record;
```

We can write a compare function, `record_compare`:

```
int record\_compare(void const *a, void  const *a)
  {  return ( ((Record *)a)->key - ((Record *)b)->key );
  }
```

Assuming that we have an `array` of `array_length Record`s suitably filled with date we can call `qsort` like this:

```
qsort( array, arraylength, sizeof(Record), record_compare);
```

Further examples of standard library and system calls that use pointers to functions may be found in Chapters 15.4 and 19.1.

Exercises

Exercise 12476

Write a program last that prints the last n lines of its text input. By default n should be 5, but your program should allow an optional argument so that

```
                        last -n
```

prints out the last n lines, where n is any integer. Your program should make the best use of available storage. (Input of text could be by reading a file specified from the command or reading a file from standard input)

Exercise 12477

Write a program that sorts a list of integers in ascending order. However if a -r flag is present on the command line your program should sort the list in descending order. (You may use any sorting routine you wish)

Exercise 12478

Write a program that reads the following structure and sorts the data by keyword using `qsort`

```
typedef struct {
        char   keyword[10];
                                                                int    other_data;
} Record;
```

Exercise 12479

An ***insertion sort*** is performed by adding values to an array one by one. The first value is simply stored at the beginning of the array. Each subsequent value is added by finding its ordered position in the array, moving data as needed to accommodate the value and inserting the value in this position.

Write a function called `insort` that performs this task and behaves in the same manner as `qsort`, ***i.e*** it can sort an array by a ***key*** value of ***any type*** and it has similar prototyping.

Dave Marshall
1/5/1999