

# Arrays

Objectives:

Having read this section you should have a good understanding of the use of arrays in C.

Advanced Data Types:

Programming in any language takes a quite significant leap forwards as soon as you learn about more advanced data types - arrays and strings of characters. In C there is also a third more general and even more powerful advanced data type - the pointer but more about that later. In this section we introduce the array, but the first question is, why bother?

There are times when we need to store a complete list of numbers or other data items. You could do this by creating as many individual variables as would be needed for the job, but this is a hard and tedious process. For example, suppose you want to read in five numbers and print them out in reverse order. You could do it the hard way as:

```
main()
{
  int a1,a2,a3,a4,a5;
  scanf("%d %d %d %d %d",&a1,&a2,&a3,&a4,&a5);
  printf("%d %d %d %d %d",a5,a4,a3,a2,a1);
}
```

Doesn't look very pretty does it, and what if the problem was to read in 100 or more values and print them in reverse order? Of course the clue to the solution is the use of the regular variable names a1, a2 and so on. What we would really like to do is to use a name like a[i] where i is a variable which specifies which particular value we are working with. This is the basic idea of an array and nearly all programming languages provide this sort of facility - only the details alter.

In the case of C you have to declare an array before you use it - in the same way you have to declare any sort of variable. For example,

```
int a[5];
```

declares an array called a with five elements. Just to confuse matters a little the first element is a[0] and the last a[4]. C programmer's always start counting at zero! Languages vary according to where they start numbering arrays. Less technical, i.e. simpler, languages start counting from 1 and more technical ones usually start counting from 0. Anyway, in the case of C you have to remember that

```
type array[size]
```

declares an array of the specified type and with size elements. The first array element is array[0] and the last is array[size-1].

Using an array, the problem of reading in and printing out a set of values in reverse order becomes simple:

```
main()
{
  int a[5];
  int i;
  for(i =0; i < 5; ++i) scanf("%d",&a[i]);
  for(i =4; i > =0; --i) printf("%d",a[i]);
}
```

Well we said simple but I have to admit that the pair of for loops looks a bit intimidating. The for loop and the array data type were more or less made for each other. The for loop can be used to generate a sequence of values to pick out and process each element in an array in turn. Once you start using arrays, for loops like:

```
for (i=0 ; i<5 ; ++i)
```

to generate values in the order 0,1,2 and so forth, and

```
for(i=4;i>=0;--i)
```

to generate values in the order 4,3,2... become very familiar.

In Dis-array:

An array of character variables is in no way different from an array of numeric variables, but programmers often like to think about them in a different way. For example, if you want to read in and reverse five characters you could use:

```
main()
{
    char a[5];
    int i;
    for(i=0; i<5; ++i) scanf("%c",&a[i]);
    for(i=4;i>=0;--i) printf("%c",a[i]);
}
```

Notice that the only difference, is the declared type of the array and the %c used to specify that the data is to be interpreted as a character in scanf and printf. The trouble with character arrays is that to use them as if they were text strings you have to remember how many characters they hold. In other words, if you declare a character array 40 elements long and store H E L L O in it you need to remember that after element 4 the array is empty. This is such a nuisance that C uses the simple convention that the end of a string of characters is marked by a null character. A null character is, as you might expect, the character with ASCII code 0. If you want to store the null character in a character variable you can use the notation \0 - but most of the time you don't have to actually use the null character. The reason is that C will automatically add a null character and store each character in a separate element when you use a string constant. A string constant is indicated by double quotes as opposed to a character constant which is indicated by a single quote. For example:

"A"

is a string constant, but

'A'

is a character constant. The difference between these two superficially similar types of text is confusing at first and the source of many errors. All you have to remember is that "A" consists of two characters, the letter A followed by \0 whereas 'A' is just the single character A. If you are familiar with other languages you might think that you could assign string constants to character arrays and work as if a string was a built-in data type. In C however the fundamental data type is the array and strings are very much grafted on. For example, if you try something like:

```
char name[40];
name="Hello"
```

it will not work. However, you can print strings using printf and read them into character arrays using scanf. For example,

```
main()
{
    static char name[40] ="hello";

    printf("%s",name);
    scanf("%s",name);
    printf("%s",name);
}
```

This program reads in the text that you type, terminating it with a null and stores it in the character array name. It then prints the

character array treating it as a string, i.e. stopping when it hits the first null string. Notice the use of the "%s" format descriptor in scanf and printf to specify that what is being printed is a string.

At this point the way that strings work and how they can be made a bit more useful and natural depends on understanding pointers which is covered in the next section.