1 Pointers

Finally, we arrive at an important stage in our study of the C programming language: Pointers. Very briefly, a pointer is a variable that holds the memory address of "something else". Exactly what kind of "something else" depends on the type of the pointer. For instance, you can have pointers to `ints`, pointers to `floats`, and pointers to almost any kind of thing C can store in memory. Let us start with a simple example, and then try to understand what happens in the program:

```
#include <stdio.h>

int main(void)
{
  int *ptr;
  int i = 54;

  ptr = &i;

  printf("The value of i is: %d\n", *ptr);

  return 0;
}
```

The first new thing here is the declaration `int *ptr;`. This defines a pointer, the name of which is `ptr`, which will be used to point to an `int`eger. The `*` is the key here, which makes `ptr` a pointer. This declaration line is read "integer pointer ptr".

The next interesting thing happens in the line `ptr = &i;`. You have seen the `&` operator as a bitwise logical operator, but that is true only when it is used as a binary operator. In its unary use (as it is used here) it is the "address of" operator. It returns the memory address of whatever C object it precedes. Therefore, here, the memory location of the variable `i` is stored in the pointer `ptr`.

Last, but not least, we have the line containing the `printf`. Here, the interesting part is the use of `*ptr` for printing. Hopefully, at this point you are aware that the address of the variable `i` is stored in `ptr`. Here, `*` is used as the "indirection operator", you can read it as "the value at". In effect, `ptr` is the address, and `*ptr` is the `int` value residing at that address, which in this program is obviously 54.

2 Pointer Arithmetic

Pointers are different from other numeric types because they actually reference memory locations. This severely limits the operations that can be performed on pointers. You can perform only the following operations on pointers:

- Add an integer to a pointer.
- Subtract an integer from a pointer.
- Increment a pointer using the increment ($++$) operator.
- Decrement a pointter using the decrement ($--$) operator.

All these amount to adding or subtracting an integer from the pointer. Let us look at the case where you add one to a pointer. When you add one to a pointer, it ends up pointing to the next object in memory. The computer calculates how many bytes the pointer must advance in order to point to the next object from the base type of the pointer. So, assuming 4-byte `int`s, if you add one to a pointer of type `int *`, the actual value of the pointer will be increased by 4, and it will end up pointing to the next `int` in memory past the one it was pointing to. If you add an integer other than one, it will advance that many positions in memory. In case of subtractions, it will fall back rather than advance.

3 The Connection between Arrays and Pointers

Arrays and pointers are closely related in C. In fact, the array variable (without an index) is a pointer to the first element of the array. Look at the following example code:

```
#include <stdio.h>

int main(void)
{
  int a[4] = {5, 7, 9, 11};
  int *b;

  b = a;

  *b = 100;
  *(b+1) = 101;
  *(b+2) = 102;
  *(b+3) = 103;

  printf("%d %d %d %d\n", a[0], a[1], a[2], a[3]);

  return 0;
}
```

Here, we define an array of four `int`s. As mentioned, the variable `a` is in fact a pointer to the start of the array. Thus, by the assignment `b = a;` we put that address into the pointer `b`. Now, we assign four different numbers to the array elements by using dereferencing on `b`. Note how we add integers to `b` to refer to other elements of the array other than the first one.

It may be surprising, but in general the following two forms are equivalent:

```
b[n]
*(b+n)
```

This means that the following code is valid:

```
#include <stdio.h>

int main(void)
```

```
{
  int a[4] = {5, 7, 9, 11};

  *a = 100;
  *(a+1) = 101;
  *(a+2) = 102;
  *(a+3) = 103;

  printf("%d %d %d %d\n", a[0], a[1], a[2], a[3]);

  return 0;
}
```

Even more interestingly, this code is also valid:

```
#include <stdio.h>

int main(void)
{
  int a[4] = {5, 7, 9, 11};
  int *b;

  b = a;

  b[0] = 100;
  b[1] = 101;
  b[2] = 102;
  b[3] = 103;

  printf("%d %d %d %d\n", a[0], a[1], a[2], a[3]);

  return 0;
}
```

So, in general, arrays and pointers are almost interchangeable. The only restriction is on array variables. You can not modify the value of an array variable, it will *always* point to the start of the array it was made for.

4 Multiple Indirection

In C, it is possible to have pointers to pointers, pointers to pointers to pointers and so on. Let us see in an example how this happens:

```
#include <stdio.h>

int main(void)
{
  int i = 10;
  int *p;
  int **ptop;
```

```
  p = &i;
  ptop = &p;
  **ptop = 20;

  printf("%d\n", i);

  return 0;
}
```

Here, `p` is a pointer to an integer, and `ptop` is a pointer to an integer pointer. First we put the address of `i` in `p`. Then we put the address of `p` in `ptop`. And then, we use `ptop` with double indirection to modify the value of `i`. It is hard to come up with a good and simple example for the use of pointers to pointers, therefore you will have to do with this one for now.

5 Connection with Strings

Since strings are arrays in C, pointers are closely related to strings. Obviously, the type of pointer that can be used with a string is a pointer to a character (`char *`). Almost all of the string library functions in C take arguments of type `char *`.

One interesting point is that it is possible to initialize a `char *` with a string constant as follows:

```
char *message = "Hello, world!";
```

When you do this, `message` becomes a pointer to the string constant. You can use this string as a read-only string. You can print it, copy it elsewhere, but you can not write into this string, because in ANSI C it is forbidden to write into string constants. (This was possible in what is called “traditional C”, but not any more.) If you want a string you can write into, you will have to define an array.

6 An Implementation of `strlen`

Here is an example implementation of the `strlen` function, which I will call `mystrlen`:

```
int mystrlen(char *s)
{
  int len=0;

  while(*s++) {
    len++;
  }

  return len;
}
```

7 Exercise 1

Write an implementation of the `strcpy` function called `mystrcpy` which copies one string onto the other. Check that it really works.

8 Exercise 2

Write an implementation of the `strcat` function called `mystrcat` which appends one string to the end of the other. Check that it works properly.