

## 10 Pointers

When a variable is declared, as with

```
double x;
```

then a memory location of 8 bytes is set up to hold the value of that variable. That memory location has an *address* which itself is just a number. A computer with, for example, 512 megabytes = 536,870,912 bytes of memory will have physical memory addresses ranging from 0 through 536,870,911. If all that memory were to be allocated to **double** variables, then all those variables would have addresses that were 8 bytes apart.

In fact, the low memory addresses are assigned to the operating system and cannot be used for storage by user programs. User programs are, however, given blocks of memory to use for storing variables. In C, if **x** is a variable declared to be a **double**, then the value of the expression **&x** is the *address* of the variable **x**.

In C, that address is an **int** or a **long** (depending on the machine and the compiler) and one can use that address in expressions.

One can also declare a variable that is itself an address: the declaration

```
double *y;
```

says that **y** is a variable that will hold the *address* of a **double** variable. If **y** is declared as above to be a variable that will hold the *address* of a **double** variable, then

```
z = *y;
```

assigns to a **double** variable **z** the value pointed to by **\*y**.

We read the declaration **double \*y** as “declare **y** to be a pointer to a double” and **\*y** in an expression to be “the **double** value pointed to by **y**.” For example, the code

```
double x,*y,z;
```

```
x = 1.0;
```

```
y = &x;
```

```
z = *y;
```

first assigns 1.0 to **x**, then assigns the address of **x** to the variable **y**, and then assigns to **z** the value (in this case 1.0) pointed to by **y**.

## Unsigned variables

At this point it is useful to introduce a new data type. When a variable is declared to be a **short**, **int**, or **long**, then the the values for that variable are taken to be represented in 2s complement notation. Essentially, for a 32-bit **int** or **long**, one uses half the representations for negative numbers and half for positive numbers, so the maximum positive value is  $2^{31} - 1$ .

It sometimes happens, though, that one does not need *any* negative numbers. For example, addresses in a computer normally run sequentially upward from 0 and are always positive. In such circumstances, it is useful to be able to get the extra bit's worth of integer range. A variable declared

```
unsigned short;
```

```
unsigned int;
```

```
unsigned long;
```

represents positive numbers only. Since a **short**, for example, is 16 bits long, this means we represent positive numbers from 0 through  $2^{16} - 1 = 65535$ . The other variables similarly represent numbers of “full” bit width instead of one bit less. These can be printed using a **%ud** or **%uld** format.