

SUMMARY OF POINTERS, ARRAYS, AND REFERENCES

By Todd Arbogast

The University of Texas at Austin

Spring 1999

Computer memory chips store *bits* of information. A bit has only two possible values, either 0 or 1; however, multiple contiguous bits can be used to differentiate more possibilities. For example, 8 bits (called a *byte*) can differentiate $2^8 = 256$ cases, enough to describe any character on a standard computer keyboard. Multiple bytes, contiguous in the memory, can be used to describe numbers, in either integer or floating point format.

In C and C++, every set of bytes in memory that are used in the program must be declared as being of some *data type*. The data type determines how the bits are interpreted to form a number or character or other object.

Each byte of memory has both a numerical location in the computer memory chip (its *address*) and the information it stores (its *value*). C and C++ allow us to access either the address or the value of a byte or contiguous set of bytes in the memory.

1. BASIC NUMERIC DATA TYPES

Ex: `<type> <name>;`

```
double x;
float y;
int i;
```

Each statement associates a C-variable `<name>` with a memory location (of an appropriate number of contiguous bytes), and declares the data `<type>` of the value. In the example above, `x` is a `double` (double-sized, long floating point number), `y` is a `float` (standard-sized floating point number), and `i` is an `int` (standard-sized integer).

In C, we access or modify the memory value through the variable name; that is, the name represents symbolically the value. We access the pointer through the `&`-operator, as in `&x`, `&y`, or `&i`.

2. BASIC STATICALLY ALLOCATED NUMERIC ARRAYS

Ex: `<type> <name>[<size>;`

```
double a[3];
float b[12];
int c[4];
```

Each statement associates a C-variable `<name>` with a series of `<size>` contiguous memory locations and declares the data `<type>` of the values. In these examples, `a` is an array of 3 `double`'s, `b` is an array of 12 `float`'s, and `c` is an array of 4 `int`'s.

We access or modify the *i*th memory value as `<name>[i]`, as in `a[i]`, `b[i]`, or `c[i]`. Note that the count starts at 0, so `i` ranges over `0, 1, 2, ..., <size> - 1`.

The `<size>` cannot be a variable; it must be fixed and known when we compile the program, and it can never be changed during execution of the program. See dynamic allocation below to circumvent this limitation.

3. BASIC NUMERIC POINTER TYPES

Ex: `<type> * <name>;`

```
double* xp;
float* yp;
int* ip;
```

Each statement associates a C-variable `<name>` with a memory location, and declares that the data type of the value is a *pointer* to some other `<type>` of data. That is, the value is interpreted as the address of another memory location, and the value of that other memory location has the given data type. In the examples above, `xp` is a pointer to a `double`, `yp` is a pointer to a `float`, and `ip` is a pointer to an `int`.

In the C-code, we can access or modify the pointer through the variable name; that is, we can access or modify the number that stores the address of the other memory location (the one we point at, not the value of the other memory location). A modification of the pointer changes which memory location we point at. Thus the name represents symbolically the pointer itself.

We access or modify the value of the memory location pointed at by *dereferencing*; that is, through `*<name>` (`*xp`, `*yp`, or `*ip` in the example above).

```
Ex: double* xp; // defines pointer (to double) xp
double y, z; // defines doubles y and z
xp = &y; // sets xp to point to y (i.e., xp = address of y)
*xp = 2.0; // sets y to 2.0 (currently y and *xp represent
// the same memory value)
xp = &z; // sets xp to point to z (i.e., xp = address of z)
*xp = 3.0; // sets z to 3.0 (currently z and *xp represent
// the same memory value)
```

4. BASIC DYNAMICALLY ALLOCATED NUMERIC ARRAYS

```
Ex: <type>* <name>;
<name> = new <type>[<size>]; // C++ only
...
delete[] <name>;

double* a;
a = new double[12];
...
delete[] a;
```

We declare the `<name>` of a pointer of the appropriate `<type>`, and later allocate contiguous space for it (i.e., we reserve a new set of `<size>` memory locations, and set the pointer to point to the first one). Subsequently, we can delete the memory allocated. In the example, `a` is a pointer to a `double`, and then it is set to point to a new set of 12 memory locations. Subsequently, these 12 locations are deleted, meaning that we will no longer access or modify them (so the computer can reuse them as it likes).

We access or modify the i th memory value as `<name>[i]`, as in `a[i]`. Again, the count starts at 0, so i ranges over $0, 1, 2, \dots, <size> - 1$.

The `<size>` can be a variable; that is, it need not be fixed or known when we compile the program. The memory is allocated during program execution to the size requested.

The syntax for C programs is different, as illustrated below.

```

Ex:  #include <stdlib.h>

      <type>* <name>;
      <name> = (<type>*) malloc(sizeof(<type>)*<size>); // C and C++
      ...
      free(<name>);

```

5. REFERENCES

```

Ex:  <type> & <name1> = <name2>; // C++ only

```

```

double y;
double & x = y;

```

In C++, we can declare that `<name1>` is equivalent to `<name2>`. This is not allowed in C. In the example, `x` and `y` are names for the same memory location.

This construction is used mainly in passing arguments to functions (see below).

6. FUNCTION ARGUMENTS: BASIC NUMERIC TYPES

Function arguments are passed by value, not by pointer. This means that the function argument is a new memory location whose value is a copy of the argument value from the function call. The following examples will illustrate this rule, and describe how to pass pointers rather than values when desired.

```

Ex:  #include <stdio.h>

      void f(int n) { n = 5; }

      void main(void) {
          int m = 2;
          f(m);
          printf("%d\n",m); // prints 2
      }

```

The function `f` above is called by `main` with the argument `m`. Its value, 2, is copied to another memory location, called `n` in the function `f`. The address of `n` and `m` are *not* the same. You can check this by adding the line

```

      printf("Address of n = %p\n",&n); to f and
      printf("Address of m = %p\n",&m); to main.

```

So, when `f` changes `n` to 5, `m` is unaffected. Note that `int n` is normally understood to create a new memory location: it does so even when this expression appears in a function's calling sequence.

If our intent is to allow `f` to change `m`, we have two options. In the first example below, we pass not the value but the pointer explicitly.

```

Ex:  #include <stdio.h>

      void f(int* n) { *n = 5; }

      void main(void) {
          int m = 2;
          f(&m);
          printf("%d\n",m); // prints 5
      }

```

Note that `n` is now a *pointer*, not an `int`. When the function is called, the number describing the address of `m` is copied into `n`, so `n` points to `m`. Since `f` changes the memory location that `n` points to (i.e., `m`), when `main` prints `m`, it is 5.

A more convenient way to achieve the above in C++ (but not in C) is to use references. What we do is *pass by reference*, as follows.

Ex: `#include <stdio.h>`

```
void f(int& n) { n = 5; } // C++ only

void main(void) {
    int m = 2;
    f(m);
    printf("%d\n",m); // prints 5
}
```

Recall from above that the expression `int& n` says that `n` is just another name for an existing memory location. When the function `f` is called, `n` becomes another name for `m`, so when `main` prints `m`, it is 5. If one were to print the pointers of `n` and `m` as mentioned above, they would now be the same.

7. FUNCTION ARGUMENTS: ARRAYS

Normally one would not want to copy an entire array, since it may be very long. Thus, both static and dynamic arrays are passed by reference.

Ex: `#include <stdio.h>`

```
void f(int n[2], double* b) {
    n[1] = 5;
    b[9] = 3.4;
}

void main(void) {
    int m[2];
    double* a = new double[10];
    m[1] = 2;
    a[9] = 0.0;
    f(m,a);
    printf("%d %g\n",m[1],a[9]); // prints 5 and 3.4
}
```

Actually, the syntax shows that the pointer `a` is passed by value; that is, the pointer value is copied, but *not* the memory locations pointed to. Although the syntax is less clear, the static array `m` is also passed by reference (i.e., as a pointer). We could change the first line of `f` to

```
void f(int* n, double* b) {
with the same results.
```