# Appendix B

# Pointers and Malloc

For a more complete treatment of this material, refer to your favorite C reference.

## B.1 Memory and Pointers

**To be blunt, most of this section should be a review. If much of the material in this section is completely new to you, then there is a good chance that you are not prepared to take S-Q. The majority of the things we'll do in this course require extensive use of pointers, and this material must be understood in order for you to complete the assignments.**

### B.1.1 Memory and Addresses

C treats memory as a large array of bytes. Every piece of memory that a program uses actually resides somewhere in this array.

The *address* of a location of memory is the index of its beginning in this "memory array". It is often useful to refer to a variable or other object by this location, instead of by its name. In fact, this is the only possible way to deal with dynamically allocated objects (discussed in section B.3.1), which do not have names in the usual sense.

A *pointer* is a C term for an address; pointers and addresses are synonymous, even though one or the other is usually used in particular idioms. (This may seem confusing at first, but after a while you won't even hear the difference.)

### B.1.2 C Syntax for Pointers

C has two operators for working with pointers.

& *Take the address of* or *get a pointer to* some location in memory.

* *Deference an address* or *follow the pointer*. Fetch the value stored at the given address.

These are both *unary* operators (i.e. they have one operand), and they come before the expression that they operate upon.

In C, pointers have specific types. The type of a pointer is that of a pointer to the type of thing that it can point to. This type is specified by following the type pointed to with an asterix (*).

```
int i;              /* i is an ordinary int.       */
int *ip;            /* ip is of type "int *".  It  */
                    /* is a pointer to an int.     */
int **ipp;          /* ipp is of type "int **"; it */
                    /* is a pointer to a pointer   */
                    /* to an int.                  */
```

Be careful, however, because the asterix in one of these types is associated with the variable, not the type. For example:

```
int i;              /* i is an ordinary int.       */
int *ip1, ip2;      /* ip1 is of type "int *", but */
                    /* ip2 is an ordinary int.     */
```

For this reason, most C programmers place the asterix in a pointer type at the start of the variable name, rather than the end of the type, in order to avoid confusion. Instead of writing "`int* ip`", which makes it look like the `int` and the asterix are somehow bound together, they would write "`int *ip`".

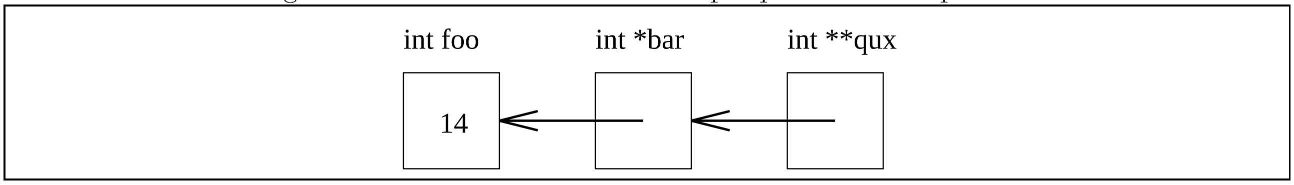
The exception to the strict typing of pointers is the *void pointer* type, which can point to something of any type.

*Pointer casts* can be used to override these types (in a manner similar to the way that type casts can be used to convert integers into doubles and vice versa). Pointer casts are generally something to avoid, except in the case of a void pointer. A `void` pointer must be cast (or assigned to a variable of a specific variable type) before it can be dereferenced.

Figure B.1: A simple pointer example.

```
int foo;         /* foo is an ordinary int. */
int *bar;        /* bar is a pointer to an int. */
int **qux;       /* qux is a pointer to a pointer to an int */

foo = 14;        /* foo gets value 14. */
bar = &foo;      /* bar points to foo. */
qux = &bar;      /* qux points to bar. */
```

Figure B.2: The result of the simple pointer example.




As an example of the use of pointers, consider the code shown in figure B.1. After this code is executed, the result is illustrated by figure B.2.

With these pointers set up, there are now several ways that we can access the value stored in the variable `foo`. We can use `foo` itself, as always; the fact that other variables point to `foo` does not limit the way that we use `foo` in any manner.

```
foo = 12;       /* assigns the value 12 to foo. */

printf ("%d\n", foo); /* prints the value of foo*/
```

We can also access the value stored in `foo` via the pointer from `bar` to `foo`:

```
*bar = 12;      /* assigns the value 12 to      */
                /* whatever bar points to.      */
printf ("%d\n", *bar); /* prints the value of   */
                /* whatever bar points to.      */
```

Since `qux` points to `bar`, and `bar` points to `foo`, we also can deref `qux` twice in order to access `foo`:

```
    **qux = 12;     /* assigns the value 12 to      */
                    /* whatever the pointer qux     */
                    /* points to points to.         */
    printf ("%d\n", **qux); /* prints the value of  */
                    /* whatever the pointer qux     */
                    /* points to points to.         */
```

### B.1.3 Call by Reference versus Call by Value

In general, C functions use *call by value* for passing their arguments: the *value* of expressions used as arguments to functions are passed to the functions, and not the expressions themselves.

For example:

```
void foo (int y)
{
    y = 5;
    printf ("y inside foo = %d\n", y);
    return ;
}

int main ()
{
    int x = 10;

    printf ("x inside main = %d\n", x);
    foo (x);
    printf ("x inside main = %d\n", x);
    return ;
}
```

In this example, when `main` calls `foo`, the value of `x`, which is 10, is passed to `foo`, and winds up in `foo`'s own copy of `y`. If `foo` changes the value of its `y`, the value of `x` inside `main` is unchanged.

When this program is run, it will produce the output:

```
x inside main = 10
y inside foo = 5
x inside main = 10
```

However, pointers allow you to sidestep call by value. Instead of passing a copy of the value to the function, you can pass a reference (or pointer) to the variable to the function. The function can then dereference the pointer, and change the variable as it pleases. This way of passing parameters is called call by reference. For example:

```
void foo (int *y)
{
    *y = 5;
    printf ("*y inside foo = %d\n", *y);
    return ;
}

int main ()
{
    int x = 10;

    printf ("x inside main = %d\n", x);
    foo (&x);
    printf ("x inside main = %d\n", x);
    return ;
}
```

This program will produce the following output:

```
x inside main = 10
*y inside foo = 5
x inside main = 5
```

## B.2 Arrays and Pointers in C

In C, there are strong relationships between pointers and arrays. For any one-dimensional array `A` and integer `x`, the following relationships are always true:

1. `&A[0] == A`

   The address of the first element of the array `A` is value of `A` itself. In other words, `A` is a pointer, it points to the first element in the array.

2. `&A[x] == A + x`

   The address of the $x$'th element of `A` is the value of `A + x`.

   This is one of the great truths (and a defining characteristic) of the C programming language. The first relationship is a special case of this more general relationship.

   This equality defines part of the nature of *pointer arithmetic*, which defines how the addition operator works when it is computing the sum of a pointer and an integer. Instead of doing ordinary arithmetic (treating `A` as though it was an ordinary integer and adding `x` to it), C takes into consideration the size (in bytes) of the type pointed to by `A`. The actual arithmetic performed (behind your back by the compiler) is to compute `A + (x * sizeof(t)`, where `t` is the type pointed to by `A`.

   For example, if `A` points to address 1000, and is a pointer to an `int`, then `A + 1` evaluates to a pointer to address 1004 on the course machines (because on the course machines, an `int` consists of 4 bytes). Similarly, if `B` points to address 2000, and is a pointer to a `char`, then `B + 3` evaluates to 2003 (because on the course machines, a `char` consists of a single byte).

   (Note that you can also subtract an integer from a pointer, which allows you to index backwards off the front of an array. For example, `A - 1` gives a pointer to the "-1'th" element of `A`. There are precious few situations where it makes any sense to do this, but it can be useful.)

3. `A[x] == *(A + x)`

   This is basically the same as the previous relationship, but adds the reassuring fact that this relationship still holds if we dereference both sides of the equality.[1]

4. `(&A[x] - &A[y]) == (x - y)`

   This relationship defines subtraction of pointers. The subtraction of two pointers to type $t$ is the number of elements of type $t$ that would fit between them.

   For example, if `A` is an array of `int`s, starting at address 5000, and `x` is 10 and `y` is 4, then then (again on the course machines, where the size of an `int` is 4) `&A[x] == 5040` and `&A[y] == 5016`. The difference between 5040 and 5016 is 24. Dividing 24 by the size of an `int` gives 6, which is indeed `x - y`.

[1] It would be strange if this was not the case, but language designers have been known to do strange things.

> Note that if the two pointers being subtracted are not of the same type, the result of the subtraction is not defined. Most C compilers will not permit you to do this, but by abusing pointer casts, you can get most compilers to let you– but when you do this, the bugs that may result are your own fault.

These relationships allow us to treat arrays like pointers, and treat pointers like arrays. Both of these treatments can be very useful, particularly when dealing with dynamically allocated memory, as we will see in the next section.

# B.3 Dynamic Memory Allocation

In many cases, it is impossible to know (or impractical to determine) when a C program is written and compiled, how much memory will be used by the program when it executes.

One approach to this problem is to have the programmer figure out, when he or she is writing the program, how much space it will require (or impose arbitrary limits on how much space the program can use), and write their code accordingly. This approach is insufficient for many common kinds of programs, however. For example, imagine how wasteful it would be if your word processor were written in such a way that it always grabbed enough memory so that it could edit an encyclopedia, even if all you ever edited with it was your grocery list.

Dynamic memory allocation is a way to defer the decision of how much memory is necessary until the program is actually running, get more if you run out, or give back memory that your program no longer needs.

## B.3.1 Malloc, Sizeof and Free

### Malloc

The `malloc` (*m*emory *alloc*ation) function is the most frequently used function for dynamically allocating memory. `malloc` has the following prototype:

```
void *malloc(unsigned int size);
```

The `size` parameter indicates the number of bytes of memory that you wish to allocate. If there is sufficient memory available, then `malloc` returns a pointer to the start of a span of memory large enough to hold `size` bytes. If sufficient memory is not available, the `malloc` returns a `NULL` pointer. (`NULL` is a constant defined as part of ANSI C, and its actual definition resides in `stdio.h`. If you try to compile a file and you get error messages about `NULL` not being defined, check to make sure that you are `#include`'ing `stdio.h`.)

Because `malloc` can return `NULL` instead of a usable pointer, your code should *always* check the return value of `malloc` in order to see whether it was successful– because if it wasn't, and your program dereferences the resulting `NULL` pointer, your program will crash. In the example code given here, I'll usually omit this check in order to keep the code short and uncluttered– but in your own code, don't neglect this check.

Also note that memory allocated by `malloc` has undefined values– there's no telling what values reside in those bytes. It is a mistake to assume that the memory is filled with zeros or some other value. Most of the time when you use `malloc`, the first thing that you will do is initialize all of the memory you allocated to some known values. (Because zero is the most common initial value, another function named `calloc` also exists in the standard library. `calloc` is similar to `malloc`, but takes different arguments; see its manual page for more info.)

### Sizeof

Calculating how many bytes are necessary to store items of different types is made simpler by the `sizeof` operator. The `sizeof` operator evaluates to the number of bytes necessary to represent a specific C type or variable. For example, on the HP machines:

```
sizeof (char)   == 1
sizeof (int)    == 4
sizeof (double) == 8
```

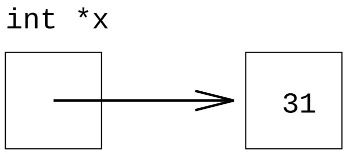
Even if you know what the size of something is, it is considered very poor style to use that number instead of relying on `sizeof`, because this makes porting your code from one platform to another difficult– and one tiny error can have strange and hard-to-diagnose results.

The return type of `malloc` is a `void` pointer. A `void` pointer may point to an object of any type, and therefore C allows a `void` pointer to be assigned to a pointer variable of any type.

For example, imagine that you wanted to allocate an `int`, and then assign 31 to this new integer:

```
int *x = malloc (sizeof (int));
*x = 31;
```

This results in the following arrangement:

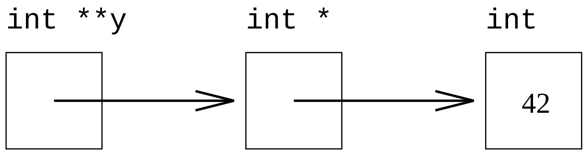



Note that this new `int` does not have a name; the only way we can refer to it is as the `int` that `x` points to. (Of course, we could create other pointers that pointed to the same place that `x` does, but in this code fragment `x` is the only pointer we have).

As a further (and more farfetched) example, consider the following:

```
int **y = malloc (sizeof (int *));
*y = malloc (sizeof (int));
**y = 42;
```

This results in the following arrangement:




Note again that the `int` pointer pointed to by `y` has no name, nor does the `int` it points to. The only way to reference this new `int` at this point is through two levels of indirection.

**Free**

When your program is finished using whatever memory it dynamically allocates, it can use the `free` function to tell the system that the memory is available again.

Take care to only `free` pointers that were returned from `malloc` (or one of its relatives); otherwise, you might free some memory that you didn't allocate, deallocating memory that might still be in use by some other part of your program. Many modern versions of `free` are intelligent enough to recognize when the program tries to free a bogus pointer, but not all of them do.

## B.4 Dynamically Allocating Arrays

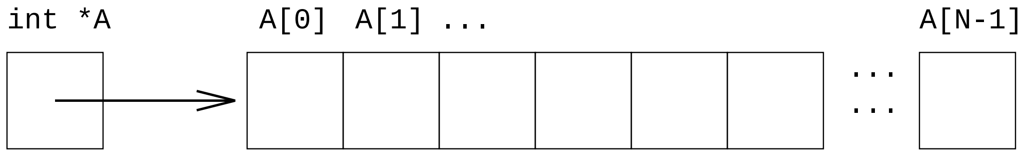
### B.4.1 One-Dimensional Arrays

In C, arrays must have their extents defined at compile-time. There's no way to postpone the definition of the size of an array until runtime. Luckily, with pointers and `malloc`, we can work around this limitation.

To allocate a one-dimensional array of length $N$ of some particular type, simply use `malloc` to allocate enough memory to hold $N$ elements of the particular type, and then use the resulting pointer as if it were an array. For example, the following code snippet allocates a block of $N$ `int`s, and then, using array notation, fills it with the values 0 through $N-1$:

```
int *A = malloc (sizeof (int) * N);
int i;

for (i = 0; i < N; i++)
    A[i] = i;
```

Pictorially, the "array" `A` might be drawn as:



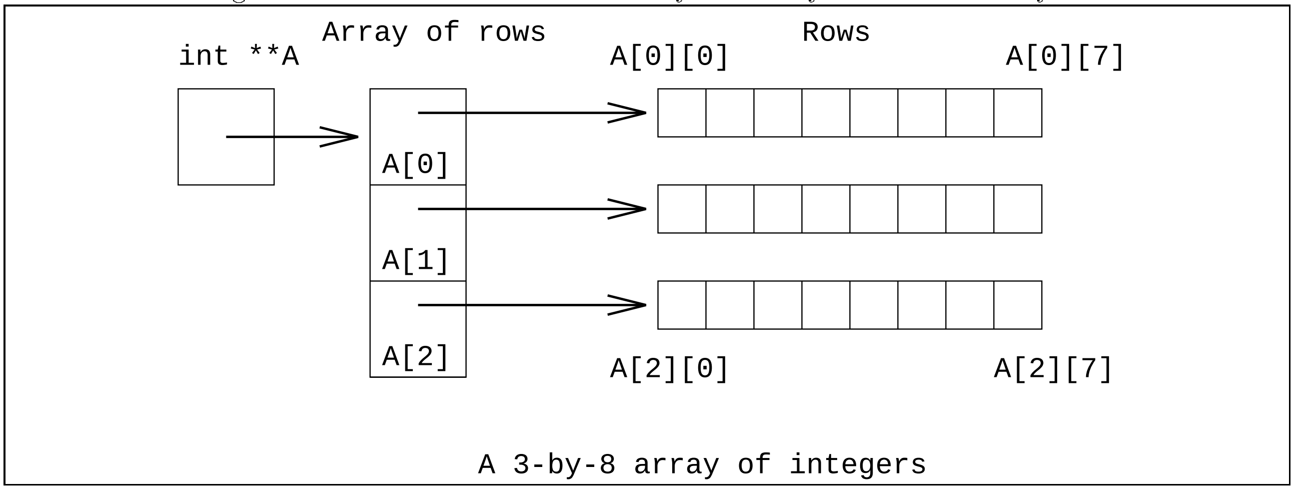

This idea is very useful for dealing with strings, which in C are represented by arrays of `char`s, terminated with a `'\0'` character. These arrays are nearly always expressed as pointers in the declaration of functions, but accessed via C's array notation. For example, here is a function that implements `strlen`:

```
int strlen (char *s)
{
    int i;

    for (i = 0; s[i] != '\0'; i++)
        ;

    return (i)
}
```

### B.4.2 Two-Dimensional Arrays

Unfortunately, the convenient relationship between pointers and arrays in C breaks down for arrays of higher dimension. Luckily, we can treat two-dimensional arrays as arrays of one-dimensional arrays. An illustration of this is shown in figure B.3, and code to allocate such an array is shown in figure B.4.

Figure B.3: A Two-Dimensional Dynamically Allocated Array




#### A Digression Into Engineering

From the perspective of software engineering, the code in figure B.4 has serious flaws. First, it doesn't do *any* error checking– it doesn't check to make sure that the number of rows and columns requested are sensible, and it doesn't check any of the values returned from `malloc` to make sure that they are not `NULL`. Next, the array returned is filled with undefined values– whatever values happened to be in the memory that `malloc` allocated for you are now the contents of this array. Finally, the array itself carries around no information about what its extents are (how many rows and columns it has). This information may be represented elsewhere in the program, but general software engineering guidelines recommend that this information be kept with the array.

Figure B.5 shows a more realistic approach, which defines a new type named `array2d_t` to represent a dynamically allocated 2-d array, and a function to create arrays and fill them with a specific initial value.

Figure B.4: Code to create an array.

```
int **arrayCreate (int rows, int cols)
{
        int i;
        int **A;

                /* Allocate the array of pointers to rows. */
        A = malloc (rows * sizeof (int *));

                /* Allocate each row. */
        for (i = 0; i < rows; i++) {
                A [i] = malloc (cols * sizeof (int));
        }

        return (A);
}
```

An illustration of the result of this code, again for allocating a 3-by-8 array, is shown in figure B.6.

Figure B.5: Better code to allocate a 2-d array.

```
typedef struct  {
        int     **elems;
        int     rows, cols;
} array2d_t;

array2d_t *array2dCreate (int rows, int cols, int initial_value)
{
        array2d_t *array;
        int i, j;

        if ((rows <= 0) || (cols <= 0))
                return (NULL);
        array = malloc (sizeof (array2d_t));
        if (array == NULL)
                return (NULL);
        array->rows = rows;
        array->cols = cols;
        array->elems = malloc (rows * sizeof (int *));
        if (array->elems == NULL) {
                free (array);
                return (NULL);
        }
        for (i = 0; i < rows; i++) {
                array->elems [i] = malloc (cols * sizeof (int));
                if (array->elems [i] == NULL) {
                        for (j = 0; j < i; j++)
                                free (array->elems [j]);
                        free (array->elems);
                        free (array);
                        return (NULL);
                }
                for (j = 0; j < cols; j++)
                        array->elems [i][j] = initial_value;
        }
        return (array);
}
```

Figure B.6: An `array2d_t` array.