Section Notes 5

CS50 — Fall 2008

Week of October 19, 2008

Contents

1 Pointers

When the language C was invented, major operating systems were still being written by hand in assembly language, i.e., machine instructions. C's designers wanted to make it practical to use a much higher-level language for efficiency critical systems software like OSes, but the language needed to give programmers the power to control data layout and manage data storage with the same precision that's available at the assembly level.

In C, unfortunately, the power to exert great control brings with it the freedom to screw up spectacularly. That's why pointers and storage management have a scary reputation with beginning students of the language. It's not that there's anything conceptually difficult about these features. It's that simple errors can corrupt the state of your program in ways that make finding the errors difficult. To avoid wasting hours debugging your code, you need to be disciplined when you create it. Make sure you understand what the pointers in your program can point to. Until you're more experienced, use lots of diagrams that show the pointer manipulations in your code. And learn to use some of the great tools that are now available to help you find problems with your use of pointers and storage. (Such as `valgrind`, which we'll talk about soon.)

Memory. As you know, your code's permanent home is in files on a disk drive. You may also have data in disk files that get read in during the running of your program. But in order for the central processor in your computer to see either code or data, it must be read into *random-access memory* (RAM), which we usually just call "memory". All of the instructions executed by the processor, all of the variables and other data in your program, whether global or on the call stack, live in memory. But what is memory?

Memory is a big array of 8-bit bytes. The instructions that the processor executes treat some parts of that big array as code, some parts as numbers, some parts as strings, and so on. For example, a variable of type `int` occupies four consecutive locations in the memory array, since `int`s are 32-bit numbers. When the processor needs to operate on that variable, e.g., to increment it by 1, it needs to know the index of the variable's first byte in memory. That index, that is, its position relative to the beginning of the memory array, expressed as a number of bytes, is called the *address* of the variable. Memory addresses are essential for locating both data and code. Processors can do nothing of interest without them.

A *pointer* in C is a data item whose value is a memory address and whose type describes the data located at that address. But why do we need pointers? In the early part of this course, we've done a good deal of programming without using explicit pointers. Why does C need to include them?

Pointers allow data structures to be shared. They make it possible for separate functions to operate on the same data locations without having to rely on global variables or arrays. They make it possible for multiple data structures to refer to exactly the same data object, so that they mirror objects in the real world, like computer networks, in which each object (or node) has multiple connections to similar objects. Pointers also make it possible to modularize data structures so that they can grow and shrink efficiently. If we only have arrays for representing ordered lists, then it can take $O(n)$ time to insert or remove a list element. As we will see later, pointers allow us to create ordered lists in which insertion and deletion of an element can be implemented in constant time, once the element's place in the list has been located.

C's very liberal rules for pointer manipulation can be risky, because they they make it hard for the compiler to recognize when we make mistakes, and hard for it to know when it's safe to optimize the code. But C's support for data sharing through pointers matches what machine-language programs do, so it's useful for us understand how to program at this level. We just need to be careful.

Creating pointers. Whenever in programming you come across a new data type that you want to understand, you should ask how objects of that type get created and what operations apply to them once you have them in hand. The simplest pointer value is the null pointer, for which the literal representation is the constant 0. By convention, we use `NULL` to stand for the null pointer instead of 0 to distinguish it from numeric zero.[1] Assigning `NULL` to a pointer location is a clean way to indicate that it points to nothing whatever. You can check whether a pointer is null just by comparing it to `NULL` (or 0) with the equality operator (==).

Pointers in C can also be derived from array names. After the declaration

```
double triple[3];
```

almost any mention of `triple` implicitly converts it to a pointer-to-`double` whose value is the address of `triple`'s first element. In effect, an array name is a pointer-valued constant. It can't appear on the left side of an assignment, the way a pointer-valued variable can, but in most other respects, an array name acts like a pointer.

C also has an address extraction operator (`&`) that applies to any expression that's valid on the left side of an assignment, such as a variable or a subscripted array expression. If `v` is a variable of type `int`, then `&v` is a pointer-to-`int` whose value is the address of `v`. If `a` is an array-of-`double`, then `&a[i]` is a pointer-to-`double` whose value is the address of the element of array `a` at index `i`.

Dereferencing. The main purpose of a pointer is to allow you to inspect or modify the location that it points to. If `p` is a pointer-to-`char` variable, then `*p` is the location pointed to by `p`, and this expression can appear either on the right side of an assignment or the left.[2] So the assignment

```
    *p = '\0';
```

would store the null terminator character into the location pointed to by `p`, which might mark the end of a string just copied into an array-of-`char`. When used in this way, the `*` is called the *dereference* operator because it goes through the pointer to access the location that it refers to. If you dereference a `NULL` pointer, the results are undefined, but highly predictable: your program will abort with a segmentation fault. (The first location in memory is never accessible.) Don't dereference `NULL`.

Because `*p` is legal on the left side of an assignment, it is valid to apply the `&` (address-of) operator to it. But `&*p` is always the same value as `p` itself, so it's not something you should write.

Syntax. Recall that when we define an array in C, we use syntax that resembles the typical way of *using* that array. The declaration

```
    double point[] = { 3.14, 2.71 };
```

is meant to be read "`point` is suitable for subscripting with the `[...]` notation, and when you do that, you get a `double`". In other words, `point` is an array-of-`double`. Similar logic motivates the syntax for other declarations in C, so to define a pointer-to-`char` variable, you write

[1] To use `NULL` in your programs, you can `#include <stddef.h>`.

[2] The binary `*` operator signifies multiplication, which means that expressions can become littered with stars when both the unary and binary forms are involved, as in `*x * *y`. Use parentheses liberally to clarify such cases.

```
char *p;
```

which is read “p can be dereferenced with the * operator (i.e., it's a pointer), and when you do that, you get a char. It is possible to define multiple names in the same declaration, but this can sometimes be confusing, because it's hard to know which parts of the declaration are shared and which aren't. The first of the following lines defines three pointers that can point to int values. The second line defines one more such pointer, plus two plain int variables:

```
int *pa, *pb, *pc;
int *px,  y,  z;
```

Mixing variable types in a single declaration is not conducive to clarity.

Casting. C is very liberal about allowing casts between pointer types. Any pointer value can be cast to any pointer type. Just like the processor can treat bytes in memory as having representing different types at different times, the C programmer can use pointer type casting to view memory contents according to whatever type is convenient at the moment. For example, when initializing all elements of a large multi-dimensional array, it can be efficient to view all of the elements as belonging to a single linear array, rather than writing nested loops to scan it. Pointer casting makes this feasible. But you should use it with caution.

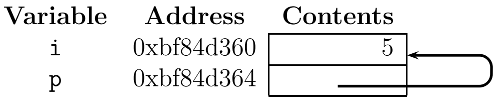
The type pointer-to-void (written void *) is special. A pointer with that type can't be dereferenced because it's not known what type of data it points to. Pointer-to-void is used as a generic pointer type. For example, a data structure might need to contain space for a pointer that sometimes points to one kind of number and sometimes to another. These pointers can only be dereferenced after being cast to a more specific pointer type. But until that type is determined, the generic type allows them to stored and passed from one function to another without obscuring the fact that they're pointers.

The constant NULL, which must never be dereferenced, but which must be assignable to any pointer location, has type pointer-to-void.

An example. Consider the following code snippet. The diagram below it shows the representation in memory after these four lines are executed:

```
int i;
int *p;          // p is now defined, but not initialized

i = 5;
p = &i;          // p now points to the location of i
```



Variable	Address	Contents
i	0xbf84d360	5
p	0xbf84d364	

A very common rookie error is to assume that because p has been defined as a pointer-to-int, it must start life pointing to a unique integer location set aside for p to point to. In fact, defining p without initialization leaves its own (pointer-valued) location containing junk, so p must *not* be used until it has been assigned a value. Once p has been set to &i, the address of i, it is ready to be dereferenced. If we now execute the assignment

```
 *p = 35;
```

p itself remains unchanged, but the value of i becomes 35.

Pointer size. Notice in the diagram above that the size of p is the same as that of i. Unlike numbers, pointers of all types typically have the same size. A 32-bit computer is one whose addresses are 32 bits wide; a 64-bit machine has 64-bit addresses, which means it can accommodate a much larger memory array.

A quiz. Test your understanding of pointer declarations and basic operations by filling in the table below to reflect effect of each statement to its left. Assume that the following declarations have been made prior to the statements:

```
int a = 3, b = 4, c = 5;
int *pa = &a, *pb = &b, *pc = &c;
```

	a	b	c	pa	pb	pc
`a = b * c;`						
`a *= c;`						
`b = *pa;`						
`pc = pa;`						
`*pb = b * c;`						
`c = (*pa) * (*pc);`						
`*pc = a * (*pb);`						

Pointer arithmetic. To round out the list of operations that can be applied to pointers, recall the close relationship between pointers and arrays. We sometimes use pointers when scanning or changing arrays because a pointer gives access to elements without having to hold onto both the array and an index into it. To move the pointer from one array element to another, we can add or subtract an integer to a pointer value.

As an example, consider the standard function `strrchr`, which returns the rightmost substring of a C string that starts with a given character. If there is no such substring, it returns `NULL`. Here is a straightforward recursive implementation that assumes its input string is not `NULL`.

```
char *
strrchr(char *str, char c)
{
    for (char *p = str; *p != ’\0’; p++)
    {
        if (*p == c)
        {
            char *better = strrchr(p + 1, c);

            if (better == NULL)
                return p;
            else
                return better;
        }
    }
    return NULL;
}
```

Only very limited arithmetic operations are defined for pointers. For example, it doesn’t make sense to add one pointer to another. You can only adjust a pointer by adding or subtracting an integer. The meaning of the operation depends on the type of the pointer being adjusted. In the `strrchr` example above, adding 1 to the pointer `p` increases the address that is its value by 1, because `p` points to a `char` value, and that occupies just one byte. But the purpose of incrementing a pointer is to advance it to a new element in a containing data structure (like an array). So if `pa` has type pointer-to-`int`, then adding 1 to `pa` actually adds 4 to the underlying address that it contains, because `sizeof(int)` is 4.

In addition to the comparison operators == and `!=`, you can use <, <=, >, and >= to compare pointer values of the same type.

Pointers and arrays. Arrays are confusing to new students of C because their special relationship with pointers is not consistent with the behavior of other compound data structures, like `struct`s (which we’ll talk about soon). So let’s review how arrays are similar to pointers and how they are different.

First the similarities. If a has been defined as an array, then in almost every context where a is mentioned, it is converted into a pointer to the first element of the array, &a[0].[3] In fact, the expression a[i] is treated exactly like *((a)+(i)) when the compiler translates your code. That's true regardless of the type of a, so when p is a pointer variable that happens to point to an array, you can use the notation p[i] to refer to the array element at position i.

A function parameter that is declared to have type array-of-T behaves *exactly* as though it had type pointer-to-T. You can assign a new value to this variable, just like any other pointer variable. You will sometimes see function main() declared like this:

```
int
main(int argc, char **argv)
{
  ...
}
```

That's because some programmers find it clearer to declare argv as a pointer-to-char* variable than to use the array-declaration notation, because the parameter is in every respect a pointer.

The key difference between array types and pointer types is that when you define an array, storage is allocated for the array's contents. When you define a pointer variable, the only storage allocated is for the pointer, not for anything that it might later point to. Consider the following definitions:

```
char s0[] = { 'x', 'y', 'z', '\0' };
char s1[] = "xyz";
char *s2  = "xyz";
```

The first two definitions have identical effect. That is, you can use a string literal as a shorthand when initializing an array-of-char. In each of those two cases, storage for an array of four characters is allocated and initialized. No storage for a pointer is allocated with s0 or s1. When those names are used, the code uses their array addresses to produce the correct pointer values. And you can never assign a new value to s0 (or s1) itself, although it's fine to assign to s1[2] (say). In contrast, you can modify s2 to make it point to some other string or to some specific character. But in this example, you can't first modify s2[2] because s2 points to a literal string constant, which you can read but not rewrite.

Call by reference. As you know, function arguments are passed "by value" in C. That means that a called function receives copies of the arguments passed to it. Its modifications to the parameter values themselves have no effect on the argument locations in the calling function.

Now that we know more about arrays, we can see that using an array name as a function argument is not quite as anomalous as it may have seemed at first. The array name is converted into a pointer to the beginning of the array, and then that pointer is passed by value to the function being called. It can't change the caller's copy of the pointer, but it can dereference its copy of the pointer to reach the same array contents seen in the caller. Thus the called function is able to modify the caller's array.

While the use of a pointer happens implicitly when the function argument is an array name, you can use the same approach to pass arguments explicitly *by reference*. Consider the following function.

```
void
swap(int *p1, int *p2)
{
    int temp;

    temp = *p1;
    *p1 = *p2;
    *p2 = temp;
}
```

By dereferencing its arguments, it is able to modify variables defined in its caller. For example, the following code snippet

[3] The exception is that sizeof(a), where a is an array, produces the size in bytes of the whole array, not the size of a pointer.

```
    int i = 0;
    int j = 1;

    swap(&i, &j);
    printf("%d %d\n", i, j);
```

prints

```
1 0
```

The caller of `swap` creates references to its local variables and passes them to `swap`, allowing it access to storage locations that would otherwise be off limits to it.

It's important never to *return* a pointer to a local variable of a subfunction back to a calling function that expects to dereference it. Local storage within a function only exists for the duration of the function call. Using such a pointer after the function has returned could lead to very difficult bugs to find.

2 Dynamic Memory Allocation

The *heap* is a pool of available memory that can be allocated *dynamically*, *i.e.,* while your program is running, and that doesn't disappear when the function in which it's allocated returns to its caller. Other memory resources that we've used so far are either *static*, meaning that the size of the storage is fixed when the program is compiled, or *local* to a particular function, so that the storage is deallocated when the function returns.

The C library provides a function called `malloc()` (short for memory allocation) that allocates a block of memory from the heap and returns a pointer to it. Whenever it needs fresh storage at run time that must outlast the current function, your program can call `malloc()`, passing it the number of bytes needed. Since the sizes of data types vary from system to system, C provides the `sizeof()` operator for determining the size of a particular type. For example, `sizeof(int)` evaluates to 4 on the servers we're using for CS 50. Note that `sizeof()` is not a function because it takes any kind of data type and returns the appropriate size, whereas a function must specify what type of argument it takes. In fact, `sizeof` computes the size of an expression at compile time.

A call to malloc might look like:

```
    int *pa = malloc(sizeof(int));

    if (pa == NULL)
    {
        printf("Error -- out of memory.\n");
        return 1;
    }
```

The return type of `malloc()` is the generic pointer type `void *`. In the snippet above, the pointer returned by `malloc()` is implicitly converted to type `int *` when it is assigned to variable `pa`.

Normally, `malloc()` returns a pointer to a block of newly allocated memory from the heap. However, if there is an error, or the heap has been exhausted, `malloc()` returns `NULL`. *Always check the pointer returned by `malloc()`.* If the returned pointer is `NULL`, you must handle this case by signaling an error. Programs that don't perform this check often dereference `NULL`, which results in a segmentation fault.

When a program has finished using a memory area in the heap, it should release the storage by calling `free()`, passing it the pointer that was originally produced by `malloc()`. `free()` only needs to be given the pointer to the memory area to be freed, not the size of the area. The C library remembers the size of each chunk allocated in the heap, so you don't have to.

There are two simple rules of thumb for safe management of heap storage:

- All memory that is `malloc`'d must later be `free`'d (but only once).
- Only memory that was produced by `malloc()` should be `free`'d.

Forgetting to free memory allocated by a program is a common programming mistake. Take care also not to apply `free` to a pointer that wasn't produced by `malloc()` (such as a locally or globally declared array in your program). And make sure not to free a pointer twice.

Be very careful to practice safe storage management, because the bugs that result from sloppiness can be hard to find. If you free a block of storage before your program has really finished using it, for example, it might be reallocated for some other purpose. So suddenly the values stored in that block may start changing and you'll have no idea why.

3 Questions on Pointers

1. What are the values of `i` and `j` after this code has executed?

```
    int i, j;
    int *ap, *bp;
    ap = &i;
    bp = ap;
    j = 10;
    *bp = j++;
```

2. What does the following program do?

```
void decrement_passed_by_value(int a);
void decrement_passed_by_reference(int *a);

int main()
{
    int a = 4;

    decrement_passed_by_value(a);
    printf("%d\n", a);

    decrement_passed_by_reference(&a);
    printf("%d\n", a);
}

void decrement_passed_by_value(int a)
{
    a--;
}

void decrement_passed_by_reference(int *a)
{
    (*a)--;
}
```

3. Write a call to `malloc()` requesting space for an array consisting of 4 `double` values. How do you access the third element of the array?