

Contents

1	Geek Humor (0:00–5:00)	2
2	Announcements (5:00–9:00)	2
3	Debugging (9:00–25:00)	2
4	Pointers (25:00–45:00)	8
4.1	Passing by Value	8
4.2	Memory Addresses and Hexadecimal	10
4.3	Passing by Reference	11

1 Geek Humor (0:00–5:00)

- Check out this [Wall Street Journal article](#) on the lamentations of Dvorak keyboard users in a world where QWERTY layouts are dominant.
- Dvorak is just one among many nerdy keyboard layouts. Others include ergonomically optimized keyboards or keyboards that have absolutely no labels at all on the keys!

2 Announcements (5:00–9:00)

- This is CS 50.
- 1 new handout.
- If you haven't already, please turn in your sensor boards from Problem Set 0.
- About grades: TFs have been trained to focus on qualitative feedback rather than quantitative feedback. Please hear us when we say that **3 out of 5 is not a 60%**! The goal of this term is for those who start off with 3's to improve toward 5's and for those who start off with 1's and 2's to improve toward 3's and 4's. Improvement is the key!
- On correctness, design, and style: correctness is simply a measure of whether or not your code does what it's supposed to, according to the specification. Design is more subjective: how *well* did you solve the problem? Your program might be 100% functional, but it could be comprised of nothing but spaghetti code. Style should be the easy part. Just make sure you have meaningful variable names, concise comments, good indentation, etc.¹

3 Debugging (9:00–25:00)

- There are more powerful tools available to you than `printf`. Among them is GDB, or GNU Debugger. Take a look at `bar.c` below:

```
/******  
 * bar.c  
 *  
 * Computer Science 50  
 * David J. Malan  
 *  
 * Offers opportunities to play with pointers with GDB.  
*****
```

¹Realize that for Problem Set 1, we graded design and style out of 2 points because we had very little time to discuss our expectations for them. From here on out, they will be graded out of 5 points.

```
*****/
#include <stdio.h>

int  foo(int n);
void bar(int m);

int
main(int argc, char *argv[])
{
    int a;
    char * s = "hello, world";
    printf("%s\n", &s[7]);
    a = 5;
    foo(a);
    return 0;
}

int
foo(int n)
{
    int b;
    b = n;
    b *= 2;
    bar(b);
    return b;
}

void
bar(int m)
{
    printf("Hi, I'm bar!\n");
}
```

This program is nothing special—it calls two functions named `foo` and `bar`—but it affords us an opportunity to walk through a program using GDB. What we'll do is compile it with a special flag that allows us to run it within the confines of GDB, like so:

```
gcc -ggdb -o bar bar.c
```

This tells GCC not only to compile our program, but to include extra information like debugging symbols. Frankly, without a debugger, this information is useless to a human and simply takes up space. But empowered with GDB, we can use it to examine the program line by line.

- First, we'll run `gdb bar`. We'll then be presented with a prompt like this:

```
(gdb)
```

From this prompt, the simplest command we can execute is `run`. When we execute this, GDB will spit out:

```
Starting program: /nfs/home/m/a/malan/cs50/4/bar
world
Hi, I'm bar!

Program exited normally.
```

Nothing very interesting. It tells us the exact path from which we're executing this program and then displays the program's normal output, telling us whether the program exited normally or not. If you wrote in non-zero return values for `main` and your program exited with one of these error codes, GDB will tell you so.

- Another quick trick: `CTRL + L` will clear the screen in Linux.
- Now let's type `break main` and hit Enter at the GDB prompt. GDB will then tell us that it has set a breakpoint at line 21 of our program. As the name implies, this will be a point at which GDB will pause execution of your program, allowing you to poke around for a bit. Now when you type `run`, you'll get:

```
Breakpoint 1, main () at bar.c:20
20      char * s = "hello, world";
```

We can see that we've stopped at the very beginning of our `main` method. Now if we type `list` we'll see the lines surrounding the current line:

```
(gdb) list
15
16      int
17      main(int argc, char *argv[])
18      {
19          int a;
20          char * s = "hello, world";
21          printf("%s\n", &s[7]);
22          a = 5;
23          foo(a);
24          return 0;
```

So why is GDB breaking on line 21 instead of 19 or 20? Line 21 seems to be the first real statement of the program, following two variable declarations.

- Let's examine the current value of `a`, which has been declared but not initialized:

```
(gdb) print a
$1 = 134514073
```

This is a good opportunity to reiterate our warning from last week that while it's fine to declare variables without initializing them, you should never use them before you yourself have assigned them a value. Clearly, the variable has some garbage inside of it because its RAM was not cleaned up after being used last.

- A sidenote: the `$1` is a temporary variable named assigned by GDB which allows you to reprint values later on, but for now let's just overlook it.
- Likewise, if we check the value of `s`, we get junk:

```
(gdb) print s
$2 = 0xb7f6edc0 "U\211?WVS??y"
```

`0x` is the beginning of a memory address, a location in RAM. Why is there junk inside of `s` at line 21? Well, the statement hasn't actually executed yet, so we need to type `next`:

```
(gdb) next
21         printf("%s\n", &s[7]);
(gdb) print s
$3 = 0x8048630 "hello, world"
```

Now, as you can see, the value we want has been stored in `s`. Recall that `char *` is just a string.

- If we type `next`, again, we'll be stepping over a `printf` statement. Let's see what gets printed:

```
(gdb) next
world
22         a = 5;
```

`world` was what printed. But what we fed in was `&s[7]`. This syntax is new, but let's see if we can take a guess. If we start counting from the left of that string, the `w` of `world` is the 7th character. Somehow we've printed a substring. More on that in a moment.

- Let's step through a few more lines:

```
(gdb) list
17     main(int argc, char *argv[])
18     {
19         int a;
20         char * s = "hello, world";
21         printf("%s\n", &s[7]);
22         a = 5;
23         foo(a);
24         return 0;
25     }
26
(gdb) print a
$4 = 134514073
(gdb) next
23         foo(a);
(gdb) print a
$5 = 5
```

If we type `next` at this point, GDB is actually going to step over `foo` to line 25. Nothing else interesting is going on with this program, so we can type `continue` to finish executing till the end (or, technically, the next breakpoint but since we've already gone past the only one, it will simply go to the end).

- What if we want to see what happens inside `foo`? Let's start over again:

```
(gdb) break main
Breakpoint 1 at 0x8048505: file bar.c, line 20.
(gdb) delete
Delete all breakpoints? (y or n) y
(gdb) break bar.c:22
Breakpoint 2 at 0x804851a: file bar.c, line 22.
```

This time, instead of breaking on the very first line, let's break on line 22. Now, we'll advance a few lines by typing `next`, but when we get to line 24, we'll type `step` instead of `next`. This will tell GDB to dig deeper into any functions that are being called on that line and to actually go line by line through them:

```
(gdb) run
Starting program: /nfs/home/a/s/asellerg/bar
world

Breakpoint 2, main () at bar.c:22
22         a = 5;
(gdb) next
```

```
23             foo(a);
(gdb) step
foo (n=5) at bar.c:31
31             b = n;
```

Here we see the first statement of the `foo` function. Let's do a quick sanity check. First, we'll `list` the lines around our current line and then we'll `print` the value of `n` and `b`:

```
(gdb) list
26
27     int
28     foo(int n)
29     {
30         int b;
31         b = n;
32         b *= 2;
33         bar(b);
34         return b;
35     }
(gdb) next
32         b *= 2;
(gdb) next
33         bar(b);
(gdb) print n
$1 = 5
(gdb) print b
$2 = 10
```

`n` takes on the value of `a` that we passed to the function and `b` is just twice that. Note that `p` is shorthand for the `print` command. Finally, we can `step` into `bar` and finish out the program:

```
(gdb) step
bar (m=10) at bar.c:40
40         printf("Hi, I'm bar!\n");
(gdb) next
Hi, I'm bar!
41     }
(gdb) continue
Continuing.
```

Program exited normally.

We used GDB here simply to step through code line by line, but don't underestimate its value as a real debugger. When you have a bug in your program, this tool is invaluable in tracking it down!

4 Pointers (25:00–45:00)

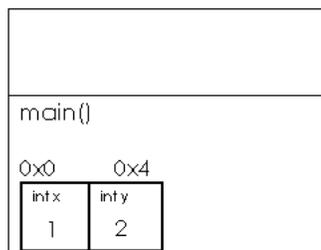
4.1 Passing by Value

- Recall from a few weeks ago our problem with `buggy3.c`:

```
void  
swap(int a, int b)  
{  
    int tmp;  
  
    tmp = a;  
    a = b;  
    b = tmp;  
}
```

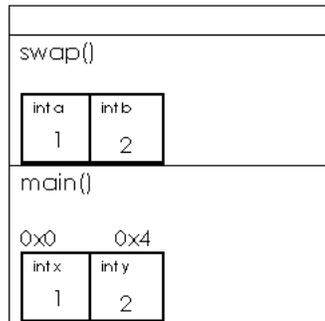
These lines of code actually functioned as they were supposed to, but the result was not what we'd hoped. This was because the values of `a` and `b` were effectively thrown away at the end of this function's execution. They were *local* to the `swap` function. What does this look like in memory? Take a look at this step-by-step representation of the stack:

1. Before we call `swap()`, the variables `x` and `y` are stored in `main`'s frame at two different memory addresses (we'll call them `0x0` and `0x4` for simplicity) with values 1 and 2.²

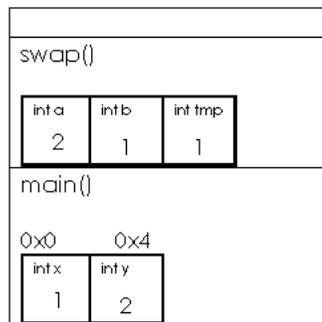


2. When we call `swap()`, copies of the variables `x` and `y` are passed to it as arguments. These become variables `a` and `b` which are stored on a new stack frame belonging to `swap()`.

²Note that in lecture, David uses memory addresses 1 and 2 along with values 7 and 8 in his example. Don't be confused!



3. Next, we declare a variable `tmp`, which will also be stored on the stack frame of `swap()`, and assign it the value of `a`. Finally, the actual swap takes place and `a` now has `b`'s value whereas `b` has `a`'s value:



4. Once `swap()` returns, however, the memory in its stack frame is freed and the stack returns to the state it was in before `swap()` was called!³
- Just as a sanity check, we can compile and run `buggy3` with GDB. As a sidenote, realize that when you run the `make` command, it's throwing in the `-ggdb` flag for you, but usually it's not a good idea to include debugging symbols in a publicly released program. These debugging symbols can actually be used to disassemble your program.
 - If we run `gdb buggy3` and `step` into the `swap` function, we can view the values of `a` and `b` which we've only been assuming so far:

```
swap (a=1, b=2) at buggy3.c:46
46      tmp = a;
(gdb) print a
$1 = 1
```

³This is not entirely true. The values of our variables might actually still be stored there, but they stand to be overwritten at any time if another function is called or our program otherwise needs to store something on the stack. Thus, the memory is not "ours" anymore!

```
(gdb) print b
$2 = 2
(gdb) print tmp
$3 = 0
```

Why do we get 0? Well, it's actually just another garbage value that happens to seem reasonable in this context. Finally, if we keep stepping and printing, we can see the swap occurs as it's supposed to, but that `x` and `y` remain unchanged at the end of the program:

```
(gdb) next
47      a = b;
(gdb) print tmp
$4 = 1
(gdb) next
48      b = tmp;
(gdb) print a
$5 = 2
(gdb) n
49      }
(gdb) print b
$6 = 1
(gdb) continue
Continuing.
Swapped!
x is 1
y is 2
```

Program exited normally.

4.2 Memory Addresses and Hexadecimal

- Although we glossed over this earlier, realize that the `int`'s we were storing on the stack had real memory locations (which we simulated as `0x0` and `0x4`, or `0x12` and `0x16` as David chose in lecture). Each `int` being 32 bits or 4 bytes, their memory addresses are then separated by 4.
- The structure of these memory addresses we're discussing is one of the reasons that certain computers can only have 2 GB of RAM. If the memory addresses themselves are 32 bits, then they can only represent numbers up to 2 billion (positive and negative). So they actually can't represent any memory location beyond that. 64-bit CPUs, in contrast, can address much more memory.
- What's the deal with the `0x` at the beginning of these memory addresses, though? These numbers are actually in *hexadecimal*. How do we count in hexadecimal? Whereas in binary you have 2 possible values for each digit

and in decimal you have 10 possible values for each digit, in hexadecimal you have 16 possible values for each digit. Check out the conversion chart below:⁴

0 _{hex} = 0 _{dec} = 0 _{oct}	0	0	0	0
1 _{hex} = 1 _{dec} = 1 _{oct}	0	0	0	1
2 _{hex} = 2 _{dec} = 2 _{oct}	0	0	1	0
3 _{hex} = 3 _{dec} = 3 _{oct}	0	0	1	1
4 _{hex} = 4 _{dec} = 4 _{oct}	0	1	0	0
5 _{hex} = 5 _{dec} = 5 _{oct}	0	1	0	1
6 _{hex} = 6 _{dec} = 6 _{oct}	0	1	1	0
7 _{hex} = 7 _{dec} = 7 _{oct}	0	1	1	1
8 _{hex} = 8 _{dec} = 10 _{oct}	1	0	0	0
9 _{hex} = 9 _{dec} = 11 _{oct}	1	0	0	1
A _{hex} = 10 _{dec} = 12 _{oct}	1	0	1	0
B _{hex} = 11 _{dec} = 13 _{oct}	1	0	1	1
C _{hex} = 12 _{dec} = 14 _{oct}	1	1	0	0
D _{hex} = 13 _{dec} = 15 _{oct}	1	1	0	1
E _{hex} = 14 _{dec} = 16 _{oct}	1	1	1	0
F _{hex} = 15 _{dec} = 17 _{oct}	1	1	1	1

Colors in HTML are often represented in hexadecimal. Generally, it's not going to be useful to you to know where your program is located in RAM, but it might be useful to pay attention to the last few digits to know where things are relative to each other. In a few weeks, when we get to the forensics problem set, you'll see memory locations written in hexadecimal. One advantage, of course, of this is that memory addresses are much shorter than they would be if we wrote them in decimal.

4.3 Passing by Reference

- So how do we go about fixing `swap`? Well, we could try to return the values after they were swapped so we wouldn't lose them. But in C you can only return a single value, so this is a bit hackish. The better solution is to pass `x` and `y` by reference rather than by value. What do we mean by this?
- Thus far, we've been passing variables by value, which is to say that we've been sending along copies of them. To pass by reference, we need to pass

⁴Source: [Wikipedia](#).

pointers to the variables. That is, we need to pass along their locations in memory so that the function is empowered to actually change the contents of that memory. Passing by reference is demonstrated in `swap.c`:

```
/*
 * swap.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Swaps two variables' values.
 *
 * Demonstrates passing by reference.
 */
#include <stdio.h>

// function prototype
void swap(int *, int *);

int
main(int argc, char *argv[])
{
    int x = 1;
    int y = 2;

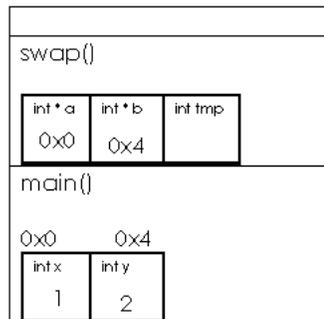
    printf("x is %d\n", x);
    printf("y is %d\n", y);
    printf("Swapping...\n");
    swap(&x, &y);
    printf("Swapped!\n");
    printf("x is %d\n", x);
    printf("y is %d\n", y);
}

/*
 * void
 * swap(int *a, int *b)
 *
 * Swap arguments' values.
 */
void
```

```
swap(int *a, int *b)
{
    int tmp;

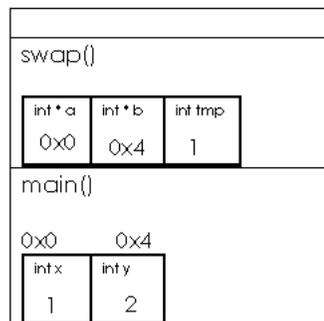
    tmp = *a;
    *a = *b;
    *b = tmp;
}
```

- Let's walk through the new `swap()` line by line while looking at the stack:⁵
 1. `int tmp;`



Here, again we're declaring `tmp` as an `int` local to the function `swap()`. However, notice now that the variables `a` and `b` are not storing values but rather memory addresses, specifically those of `x` and `y`! These are pointers to `x` and `y`. When called upon, our function can access and change `x` and `y` directly.

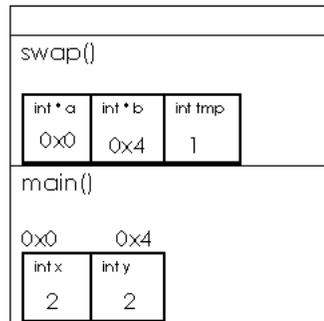
2. `tmp = *a;`



⁵Note that in lecture, David uses the memory addresses `0x12` and `0x16`, but we're using `0x0` and `0x4` here. The starting points of both sets are arbitrary—only the difference between them, namely 4 bytes, is significant.

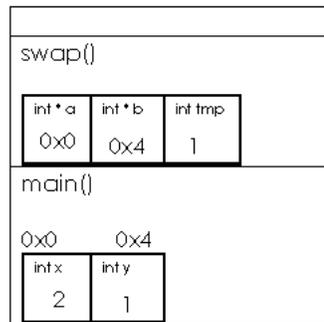
Notice the asterisk or star in front of `a`. This is the dereferencing operator. The whole line reads “assign to `tmp` whatever is stored in memory at location `a`.” Once we’ve done that, `tmp` stores the value 1, as shown in the diagram.

3. `*a = *b;`



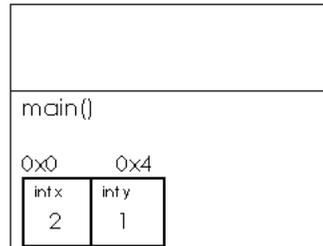
This line is a little harder to follow, but pay attention to the diagram. We’re saying “assign to the memory at location `a` whatever is stored in memory at location `b`.” This is the first part of the swap! We’ve actually modified the memory of `main`!

4. `*b = tmp;`



And the swap is complete! Note that we don’t use a star in front of `tmp` in this line. That’s because we don’t care about where `tmp` is stored, we’re only using it to temporarily hold a value. We don’t care what happens to it after `swap()` returns.

5. `[return;]`



Finally, `swap()` returns (albeit not explicitly in the code since its type is `void`). Unlike in `buggy3.c`, however, the values of `x` and `y` have actually been swapped!

Realize that many modern programming languages don't allow this kind of control over memory because it can be dangerous. We'll demonstrate in lecture how passing around memory addresses can be exploited.

- One other change we've made to `main` in `swap.c` is to pass `&x` and `&y` as opposed to `x` and `y` as we did in `buggy3.c`. The `&` operator asks for the memory location of whatever comes after it—you can think of it as the opposite of the dereferencing operator. So, as we said, we're passing the memory locations of `x` and `y` to `swap`.
- The declaration of `swap` has also changed. We specify its parameters as `int *` as opposed to just `int`. This means that its parameters are pointers to `int`'s, not just `verb/int`'s.
- Where have we seen the `*` operator before? Recall `char *`, which is synonymous with our string datatype.
- Pointers are often represented in diagrams as an arrow pointing to a box representing a chunk of memory with some value stored in it. When you declare a variable, you are getting one of these chunks of memory with a something unknown stored in it (often represented as a question mark). Check out the [tutorial](#) at How Stuff Works for a good visualization of pointers (the same that David uses in lecture).
- Let's say we have a chunk of memory after declaring `int i`:



- Now let's declare a pointer to an `int` and name it `p`. If we write `p = i`, what will we get? Nothing. We'll get a memory address of 3 which is likely not memory that we own. In fact, we'll get a warning from the compiler that assignment is between two different variable types. What we really should write is `p = &i`. Now if we say that `i` is at memory address `0x123`, our picture looks like the following:

p 0x123 i 3

- A less confusing way of representing pointers (rather than using arbitrarily picked memory addresses, is to use an actual arrow like so:

p 0x123 → i 3

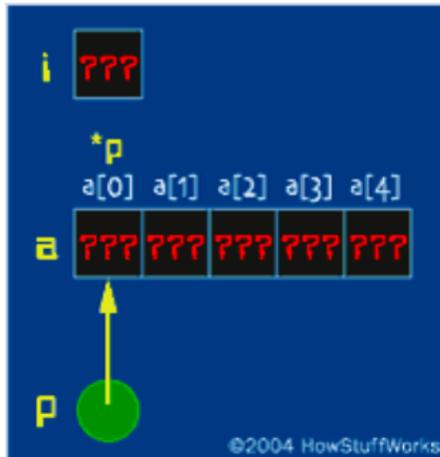
- Now, if we write `*p = 5`, then we get the following:

p 0x123 → i 5

- As it turns out, arrays are also pointers. For example, we might write:

```
int i;  
int a[5];  
int *p = a;
```

We can represent this pictorially like so:



`p` is now a pointer to the first element of `a`. This is, in fact, all that an array is. We know that all of the elements of an array are stored contiguously in memory, so once we know the first, we can calculate all the rest.

- Let's say we were to write `printf('%d', p)`, what would it print? Well, it would actually print out the address of `p` since. If we wrote `printf('%d', *p)`, instead, we'd get the actual value at the first index of array `a`.
- So from now on, we're dropping the training wheels. When we talk about strings, we're going to talk about `char *`. Now let's take a look at a program that purportedly compares two strings but actually fails in doing so:

```
/*
 * compare1.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Tries (and fails) to compare two strings.
 *
 * Demonstrates strings as pointers to arrays.
 */

#include <cs50.h>
#include <stdio.h>
#include <string.h>

int
main(int argc, char *argv[])
{
    // get line of text
    printf("Say something: ");
    char *s1 = GetString();

    // get another line of text
    printf("Say something: ");
    char *s2 = GetString();

    // try (and fail) to compare strings
    if (s1 == s2)
        printf("You typed the same thing!\n");
    else
        printf("You typed different things!\n");
}
```

First we're asking the user for two strings and then storing them. But how exactly is `GetString()` returning an entire string? Well, it's not. In fact, it's returning a pointer to the first `char`. How do we know where the end of the string is? As with an array, the string is a series of characters which are stored contiguously in memory. And the null terminator (`\0`) marks the end of the string. Recall the `strlen` function. If we think about how that might be implemented, we would probably guess that it simply iterates over all the characters of the string until it reaches the null terminator.

- What happens when we compile and run `compare1`? If we type `foo` for each of the strings, we get the message that we have different strings.

What's really going on? Each time we call `GetString()`, it's returning a pointer to the first `char` of the string—in other words, a memory location. But if we're getting two different strings, they must be stored at two different memory locations. And, in fact, when we ask if `s1 == s2`, that's what we're really comparing—the memory locations of the two strings, which will never be equal.

- The working version of this program is `compare2.c`:

```
/******  
 * compare2.c  
 *  
 * Computer Science 50  
 * David J. Malan  
 *  
 * Compares two strings.  
 *  
 * Demonstrates strings as pointers to arrays.  
*****/  
  
#include <cs50.h>  
#include <stdio.h>  
#include <string.h>  
  
int  
main(int argc, char *argv[])  
{  
    // get line of text  
    printf("Say something: ");  
    char *s1 = GetString();  
  
    // get another line of text  
    printf("Say something: ");  
    char *s2 = GetString();  
  
    // try to compare strings  
    if (s1 != NULL && s2 != NULL)  
    {  
        if (!strcmp(s1, s2))  
            printf("You typed the same thing!\n");  
        else  
            printf("You typed different things!\n");  
    }  
}
```

In a way, this is a bit of cheating. We're just calling a pre-made function.

But if we had to implement `strcmp` ourselves, how would we go about it? We'd have to walk through each character one by one in both strings. If the two characters are ever different, then return false. And if you reach the null terminator of both strings at the same time, then return true.

- There's also a sanity check in `compare2`. Only if `GetString()` doesn't return `NULL`, the sentinel value, does the string comparison proceed.
- What gets returned by `strcmp()` if two strings are equal? 0. And it returns a positive or a negative depending on whether one string is shorter or longer than the other.