Computer Science 50
Fall 2008
Scribe Notes

Week 4 Wednesday: October 15, 2008
Andrew Sellergren, Anjuli Kannan

Contents

1 Announcements (0:00–11:00) 2

2 Pointers (11:00–62:00) 2
2.1 Passing By Value 2
2.2 Passing By Reference 4
2.3 Pointer Arithmetic 9

3 Dynamic Memory Allocation (62:00–75:00) 9

1 Announcements (0:00–11:00)

- Computers are not smarter than humans![1] Read about the results of the latest attempt to pass the Turing test.
- 339 Games of 15![2]
- "Lunch with David" on Fridays: e-mail `rsvp@cs50.net` if you'd like to join.
- Use the Bulletin Board! Your question might already be answered, or, if not, it *will* be answered by a member of the staff or perhaps even a fellow student.
- Thanks to Amazon! The next problem sets will be hosted by their EC2 service.
- Thanks to Microsoft![3] Check out all the software which is yours to download and use through the MSDN Academic Alliance.
- 1 new handout.
- 6 new lolcansus.

2 Pointers (11:00–62:00)

2.1 Passing By Value

- Why doesn't this function in `buggy3.c` work as we might expect?

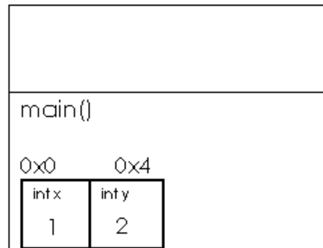
```
void
swap(int a, int b)
{
    int tmp;

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

  The variables declared inside `swap()` (e.g. `tmp`, `a`, and `b`) are *local* to that function. This means that they are stored on the stack and as soon as `swap()` returns, they will effectively disappear.

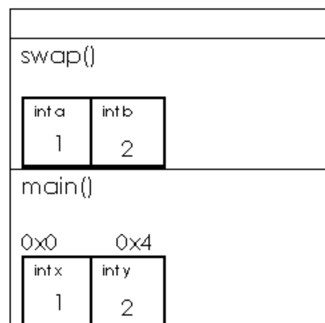
- What would this look like in memory? Take a look at this step-by-step representation of the stack frame in `buggy3.c`:

[1] Yet. . .
[2] Thank you, Mrs. Malan! Sorry, Connecticut, you'll have to restock.
[3] Tablets. We has them.

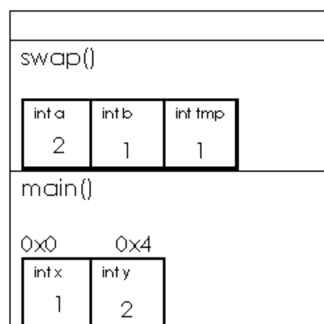
1. Before we call `swap()`, the variables `x` and `y` are stored in `main`'s frame at two different memory addresses (we'll call them `0x0` and `0x4` for simplicity) with values 1 and 2.[4]



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()`.



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`.



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

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![5]

2.2 Passing By Reference

- How do we fix this? We need to pass by *reference* as opposed to *value*. Passing by reference is a way of telling a function where in memory a variable is stored. Thus, when it accesses and changes it, the change lasts even after the function has returned. Check out `swap.c` to see what we're talking about:

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

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

  Notice the stars in front of `a` and `b` in our function definition. These are not `int`'s but rather *pointers* to `int`'s. That is, they are addresses in memory, so they point to where an `int` is stored.

- How do we pass by reference? Notice the one syntax change in `main()` between `buggy3.c` and `swap.c`:

```
swap(&x, &y);
```

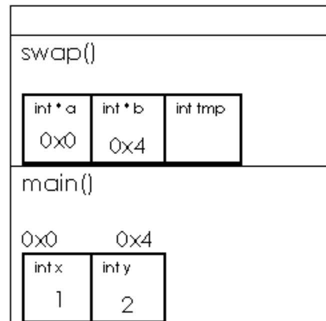
  The ampersand (`&`) is the "address of" operator, meaning the arguments to `swap()` are now the addresses of `x` and `y`, instead of simply their values. Now, we're effectively passing the variables themselves rather than copies of those variables.

- What are the advantages and disadvantages of this? The advantage is greater control for the programmer. The disadvantage is potential exploitation since there is very little to stop someone from accessing *any* address in memory. This is part of the reason why these low-level details are obscured in higher-level programming languages such as Java.

- Let's walk through the new `swap()` line by line while looking at the stack:

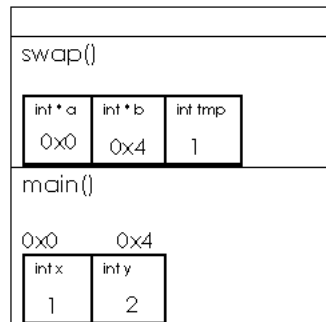
  1. `int tmp;`

[5] This is not entirely true. The values of 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!



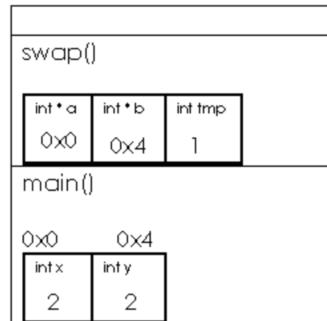
Here, again we're declaring `tmp` as an `int` local to the function `swap()`. However, notice now that the variables `a` and `b` are not storing values but rather memory addresses, specifically those of `x` and `y`! When called upon, they can tell where `x` and `y` are located in memory so that our function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

2. `tmp = *a;`



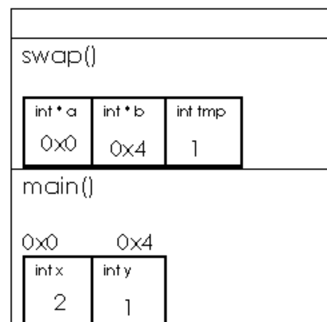
Notice the star in front of `a`. This is the dereferencing operator. The whole line reads "assign to `tmp` whatever is stored in memory at `a`." Once we've done that, `tmp` stores the value 1, as shown in the diagram.

3. `*a = *b;`



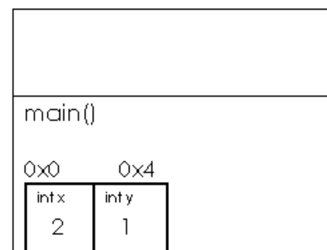
This line is a little harder to follow, but pay attention to the diagram. We're saying "assign to the memory at `a` whatever is stored in memory at `b`." This is the first part of the swap! We've actually modified the memory of `main`!

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!

- What if we were to write `tmp = a;` when we've passed `a` as a pointer? When we compile, we'll get an error which says "assignment makes integer from pointer without a cast." This is a cryptic warning which means that we have a type mismatch in our variable assignment. We could change it to `tmp = (int) a;` to make this warning go away, but this would be forcibly converting a memory address to an `int`, which we probably don't want either.

- 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).

- Remember, we've been using pointers all along. Every time you've declared `main` you've typed `char *argv[]` as one of its arguments. Turns out that an array is actually a pointer as well. When you declare an array, it returns a pointer to the first chunk of memory where that array is stored. If we declare an array `int a[5]` and then a pointer which we initialize as `int *p = a`, then we can access the first `int` stored in the array by writing either `a[0]` or `*p`.

- Keep in mind the implications of this: if only the *start* of the array is stored, then it's up to you to remember where the *end* of the array is stored! The program `compare1.c` is an example of using pointers incorrectly:

```
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");
}
```

What is a `char *` exactly? Well, it's the same thing as a `string`. For the first few weeks, we defined our own type called `string` so that you wouldn't have to worry about what a pointer was! So what does `GetString()` actually return, then? A pointer to the first character!

- So what's wrong with this program then? Take a look at condition `s1 == s2`. What are we actually comparing? Well, `s1` and `s2` are actually just storing memory addresses. If we compare them, they will never be equal, even if the strings they store are the same. Try compiling and running `compare1.c`. Even if you type the same input twice in a row, the output is "You typed different things!"
- How should we actually go about comparing strings? Well, we're going to have to compare them one character at a time. So we'll write some kind of loop that steps through each string until it hits the `NULL` terminator in one of the strings. Take a look at `compare2.c` which uses a function called `strcmp()`:

```
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");
    }
}
```

If we look at the `man` page of `strcmp()`, we see that it takes two `const char*`'s as arguments. What does the `const` mean? What you would expect: it means that the `char *`'s will not be modified by the function. This is a safety check: now that we're passing by reference as opposed to value, we don't want our memory to be modified without our consent!

- Notice that `strcmp()` returns a positive value, a negative value, or 0. These correspond to greater than, less than, or equal to, which might come in handy if you're trying to sort, for example. In `compare2.c`, we check simply if `strcmp()` returns non-zero (with the ! being the unary NOT operator).

- If we pass nothing to `compare2`, then no message is printed. This is because the outer condition `(s1 != NULL...)` is false in this case. What does `GetString()` return when it fails? It returns `NULL`, which is analogous to returning false for a function with type `int`.

2.3 Pointer Arithmetic

- What other syntactic tricks can we do with pointers? Because arrays are stored as contiguous chunks of memory, we can access them using pointer arithmetic. To get the second element of an array called `foo[]`, for example, we could write either `foo[1]` or `*(foo+1)`. Check out `pointers1.c` to see how this works.

- Moving on to `pointers2.c`:

```
int
main(int argc, char *argv[])
{
    int numbers[] = {1, 2, 3, 4, 5};

    printf("Size of array is %d.\n", sizeof(numbers));
    printf("Size of each element is %d.\n", sizeof(numbers[0]));
    for (int i = 0, n = sizeof(numbers) / sizeof(numbers[0]); i < n; i++)
        printf("%d\n", *(numbers+i));
}
```

  Note that `sizeof(numbers) / sizeof(numbers[0])` is a trick to determine the number of elements in an array. However, ignore it for a moment and focus on the pointer arithmetic. Now we have an array of `int`'s and yet we're still only adding 1 as an offset each time we access the array. If this 1 corresponds to a byte, won't we actually the first 5 bytes of the array rather than the 5 elements? Turns out the compiler can actually figure out the size of each element and increment by that size each time, so the program works as we'd hoped.

3 Dynamic Memory Allocation (62:00–75:00)

- So far, we've only been able to allocate memory by declaring a variable which is stored on the stack. What's the problem with this? If that stack frame disappears, so does the variable. We get around this in the Game of Fifteen by using a global variable, which isn't stored on the stack.

Generally, these are frowned upon, but for the game board it makes sense to have a variable which is stored somewhat permanently.

- What if we don't know how much space we'll need to store something, say, a name in a database? One of the advantages of C is that it allows us to ask for memory whenever we want it. We do this using a function called `malloc()`.
- We've actually been using `malloc()` all along. The function `GetString()` begins by asking for memory to store a string. In fact, this memory is actually never explicitly freed, so it results in what's called a *memory leak*. Over time, this unfreed memory can build up and cause your computer to slow down. Good programming practice is to explicitly free any memory we've allocated by making a call to `free()`.
- To see how `free()` is used, take a look at `copy1.c` and `copy2.c`. Both intend to copy a string and capitalize its first letter, but the former actually capitalizes both the input and the output. What's the problem? Simply assigning a string to a new pointer, as in the line `char *s2 = s1;` will actually initialize a pointer to the *same* memory storing the first string. Thus, if you change one, you change both.
- A correct implementation of copying a string is achieved in `copy2.c`:[6]

```
int
main(int argc, char *argv[])
{
    // get line of text
    printf("Say something: ");
    char *s1 = GetString();
    if (s1 == NULL)
        return 1;

    // allocate enough space for copy
    char *s2 = malloc((strlen(s1) + 1) * sizeof(char));
    if (s2 == NULL)
        return 1;

    // copy string
    int n = strlen(s1);
    for (int i = 0; i < n; i++)
        s2[i] = s1[i];
    s2[n] = '\0';

    // change copy
```

[6]Note that this code has fixed the bugs which David got called out for during lecture. Tsk tsk.

```
    printf("Capitalizing copy...\n");
    if (strlen(s2) > 0)
        s2[0] = toupper(s2[0]);

    // print original and copy
    printf("Original: %s\n", s1);
    printf("Copy:     %s\n", s2);

    // free memory
    free(s1);
    free(s2);
}
```

Here, using `malloc()` we've allocated separate memory for the copy. We've specified the amount of memory we'll need as `(strlen(s1) + 1) * sizeof(char)`. The `+ 1` is necessary so that we have room for the NULL terminator. We also make sure to check the return value of `malloc()`. If we don't get the memory we asked for, we want to be sure we don't try to store anything there!

- Know that `malloc()` allocates memory on the heap as opposed to the stack!
- Anticlimactic conclusion!