APS105: Absolutely Official Pointers Factsheet

Daniel Zingaro

February 16, 2010

Instructions

Read once a day until April 2010. Then recycle.

Here is an outline of the most important facts and confusions about pointers. The exercises should be attempted as they are reached in the handout.

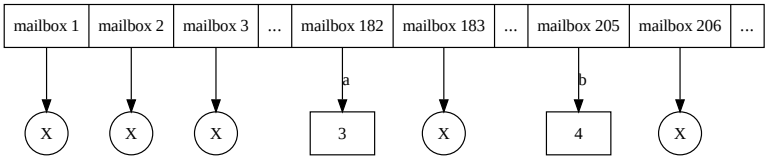
Fact 1: All Variables Have Addresses

You know when we say something like

```
int i;
i = 5;
```

? One important property of `i` is its value: it has value 5. But another important property is the variable's location, or address. The address is some value, like `25843` or `9122212`. We generally don't care what the address is, because as a raw address it is meaningless to us.

Think of memory as a bunch of mailboxes, each one having a different address. So, we have a mailbox for address 1, a mailbox for address 2, a mailbox for address 3, and so on. Each mailbox can hold one value. In the following diagram, mailbox (address) 182 is where `a` is stored, and mailbox 205 is where `b` is stored. These are the variables' addresses. If we were able to ask, "what is the address of `a`?", C would respond "182".



Fact 2: & Gives us the Address of a Variable

Let's say I declare a variable `a`:

```
int a = 3;
```

and I want to know the address of that variable. (In the above diagram, I want C to tell me "182".) How do I do this? I use the `&` operator, and say `&a`.

`&a` means "give me the address of `a`."

Even though an address looks like an integer (like 182), it's not a regular integer. It has special meaning as an address, and so we cannot say something like:

```
int myAddr = &a;
```

instead, we have special variables called pointers (stay with me here . . .). A pointer variable is a variable that holds an address. Just like integer variables hold integers and float variables hold floats, pointer variables hold addresses of variables. To declare a pointer variable, we include a `*` in front of the variable's name:

```
int *p;
```

Read this as "`p` holds the address where I can find an integer". How can I initialize `p`? I require the address of an integer — and we know how to get one of those. We use `&` on an integer variable:

```
int a = 3;
int *p = &a;
```

If `a` is stored at address 182, then `p` will have the value 182. We say that `p` "points to" `a`. `p` is the address of where I can find an integer. What that means is that I can find an integer like this:

- 1. Get the value of `p`. It's 182.
- 2. Go to memory address 182, and get the value there. The value is 3.

The question is: how do I "go to address 182"?

Fact 3: * Tells us What is Stored at an Address

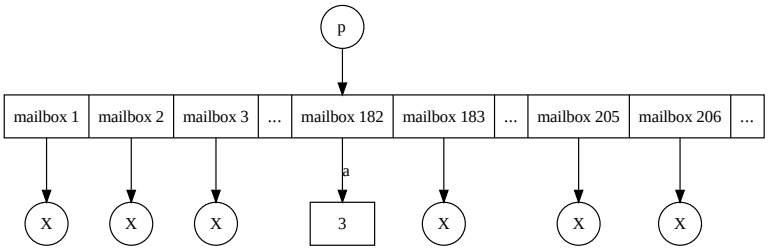
So, I have variable `p`, whose value is the address of where I can find an integer. Let's say the value of `p` is 182. I want to "follow the pointer" to address 182, and fetch the value stored there. I do this with the `*` operator.

`*p` means "get the address stored in `p` and tell me what is found at that address." (Or: "give me whatever is pointed to by `p`.")

So, `*p` means, "what is at address 182?"; the answer is 3. I can therefore print 3 in two ways, using variable `a` directly, and following `a's` address stored in `p`:

```
int a = 3;
int *p = &a;
printf ("%d\n", a);
printf ("%d\n", *p);
```

Here is the relationship between `a` and `p`.



Exercise 1

What value is printed by the following code? Be sure you understand the assignment `c = b`: it is copying the value of `b` into `c`. The value of `b` is the address of `a`.

```
int a = 3;
int *b;
int *c;
b = &a;
c = b;
printf ("%d\n", *c);
```

Fact 4: * can Store a Value in an Address

If we use `*` on the left-hand-side of an assignment statement, we can put a value in the address stored in a pointer variable. Here's an example.

```
int a = 3;
int *p = &a;
*p = 19;
```

Prior to the last line executing, you will agree that `a` and `*p` have the same value of 3. `a` and `*p` are therefore two names that refer to the same thing. When we say `*p = 19`, it is the same as `a = 19`. In other words, `*p = 19` stores 19 in `a`'s memory address. After this, we can print the value of `a` to see that it really has changed to 19. Try it!

Fact 5: Pointers let Functions Change Variables

Let's say we're in `main` and we want to call a function that changes our variable `i`:

```
int main(void) {
  int i = 18;
  /*call function to change value of i*/
  change (...);
  ...
}
```

We cannot pass `i` itself to a function that expects to change `main`'s `i`. If we try:

```
change (i);
```

then `change` is simply getting the value 18. `change` has no idea where the 18 came from; it does not know the address of that 18. It cannot possibly change the value stored in `i`.

Instead, let's pass the address of that 18:

```
change (&i);
```

Now, `change` is getting an address, and at that address it will find an integer. To define `change` properly, we must indicate that it is receiving the address where an integer can be found (i.e. a pointer to an integer). Here, we'll write `change` so that it sets the variable at the specified address to the value 42:

```
void change (int *p) {
  *p = 42;
}
```

Exercise 2

With this definition of `change`, fill in the call to `change` below so that `main`'s `i` gets value 42. Think carefully about what `change` is expecting you to pass: it wants the address of an integer, not an integer.

```
int main(void) {
  int i = 18;
  /*call function to change value of i*/
  change (...);
  ...
}
```

Recipe 1: Writing a Function to Change a Variable

To write a function that modifies a variable that you pass to it:

- 1. For each "outside" variable you want to modify in the function, put a `*` in front of its name in the function definition
  - `void change (int *p)`
- 2. For each such variable `p`, use `*p` when storing or retrieving the data at the address stored in `p`.

Exercise 3

Write a function that takes two parameters: `a`, a pointer to an integer, and `b`, an integer. Store the value of `b` at the address pointed to by `a`. Then, write a `main` function that properly calls your function, and demonstrates that it works.

Fact 6: All Variables Have a Type

If we say `int i;`, `i` can store integers. The type of `i` is `int`. Similarly, if we say `int *j;`, `j` can store addresses of integers (i.e. pointers to integers). The type of `j` is therefore `int *` or "pointer to `int`". The **only** values we can store in `j` are addresses of integers. In `j`, we **cannot** store integer values like `5`, float values like `8.2`, memory addresses of float values like `&f` (assuming `f` is a `float`), etc.

Confusion 1: Declaring Pointers

Both of these declarations are the same:

```
int *i;
int* i;
```

You'll see both in real C code. The position of the `*` does not matter: both declarations make `i` have type "pointer to `int`". However, the declarations tend to evoke different interpretations (that of course result in the same correct conclusion):

- `int *i` looks like we are defining `*i` as an `int`. If `*i` is an `int`, it means that when we follow the address stored in `i`, we get an `int`. The type of `i` must therefore be "pointer to `int`", or `int *`.
- `int* i` makes it look like we are defining `i` to have type `int *` (i.e. "pointer to `int`"). This is exactly the same interpretation as we arrived at for `int *i`.

Confusion 2: Should I use & or *??

- Use `&` when you have a variable and you want its address
- Use `*` when you have a pointer variable, and you want to access what it points to
- Use `*` in the first line of a function definition for each pointer parameter you want to receive
- Never use `&` in the first line of a function definition

Exercise 4

Here is a function that takes two pointers to integers, and returns the sum of the values that they point to.

```
int sum (int *a, int *b) {
  return (*a + *b);
```

- A. Why do we return `*a + *b` rather than `a + b`?
- B. Write a `main` function that declares two integer variables and then uses `sum` to calculate their sum. Print the sum with `printf` to test that everything is working properly.

Confusion 3: scanf and &

It's not always true that a `&` is required for a `scanf`. If we have a pointer already, we do not use `&`:

```
int a;
int *p = &a;
scanf ("%d", p);
```

If we said `&p` rather than `p`, we would be passing a pointer to a pointer to an integer. This is not what `scanf` is expecting: it is expecting a pointer to an integer, which is exactly what `p` on its own is.

Confusion 4: Initializing Pointers

We know that all variables must be initialized before used — if we do the following, we'll get garbage output:

```
int i;
printf ("Garbage: %d\n", i);
```

The situation is similar with pointers. If we just declare a pointer without first initializing it:

```
int *p;
```

then its value is going to be garbage. In other words, it contains some arbitrary address. If we try to store something there (like `*p = 1850;`), we're going to overwrite some arbitrary location in memory. It might make our program crash, or it might corrupt our data, or it might just seem to work (which is most dangerous of all!).

Confusion 5: Pointer Parameter or Return Value?

Let's say you want to write a function that gives you a random number between 1 and 6. There are two ways to do it:

- 1. Write a function that takes a pointer to an integer, and puts a random number in the variable pointed to by the pointer. The function does not return anything (e.g. `void` return).
- 2. Write a function that takes no parameters, and returns an integer.

For this particular function, the second choice is more natural: we can directly use it in a `printf` statement, and we can avoid pointers altogether. Here is a sample program that defines both of these functions and shows how they are called.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void func1 (int *i);
int func2 ();

void func1 (int *i) {
  *i = rand() % 6 + 1;
}

int func2 () {
  return (rand() % 6 + 1);
}

int main(void) {
  int randNum1, randNum2;
  srand ((unsigned)time (0));
  func1 (&randNum1);
  printf ("func1 gave me %d\n", randNum1);
  randNum2 = func2();
  printf ("func2 gave me %d\n", randNum2);
  printf ("I can also call func2 directly to get %d\n", func2());
  return 0;
}
```

For a function that swaps two values, or generates ten random numbers and puts them in ten different variables, we have no choice but to use pointers. A function can return only one value.