Printable View of: Week 3: Pointers and Accessing the command line

Print Save to File

File: pointers - the very basics

pointers - the very basics

Assignment 2

What is a *pointer*?

I hate to destroy all the mystery and intrigue, but a *pointer* is just another variable. It is only special because it contains an address, but an address is just like an int, so there really is no big deal! Watch:

```
int n;
n=6;
```

Above we declared a variable type *int* and put a value in it. It is no different with pointers:

```
int* pn;
pn=&n;
```

See? We declared a var type *int** and put a value in it.

VERY IMPORTANT STUFF HERE!!!

When you are declaring a pointer variable, remember that the * is part of the type name. In the above examples we declared an "int" and an "int*". These are different types. Because you can move the * around in the declaration:

```
int* pn;
int * pn;
int *pn;
```

are all the same. They all make a variable called "pn" of type "int*". This is cause for much confusion with fledgeling programmers. Please try to stick to the syntax in the example. If you attach the * to the base type, it helps you keep it clear in your mind that the type is "int*".

But, there is a problem. if you do

```
int* p1,p2,p3;
```

then only `p1` is a pointer. `p2` and `p3` are regular int variables. Yuc. So, if you are going to declare more than one pointer on one line, you need to do:

```
int *p1,*p2,*p3;
```

Pretty annoying, isn't it? But, that is how the compiler needs to have it.

About the * and the &

So far the * is just part of a pointer's type name. We'll see one other use for the * later.

the & is actually pretty simple to understand. Just read it "address of". Now look back at the assignment statement in our example. It says "pn gets the address of n". Could that be any more clear?! :)

So, the & just gives you the address of the variable that it precedes.

Assigning and accessing *referenced* values

It is very important that you master the vocabulary here. Pay close attention to how the terms are used and try to use them correctly yourself. A *referenced* variable is one that is accessed with a pointer. Usually a pointer contains the address of a variable that doesn't have a name, because it was created with a malloc( ) (we will discuss malloc( ) later) but in our examples here ther *referenced* variable will also have a name.

There are two different assignments to consider:

`pn=&n`

this puts the address of the integer variable "n" into the pointer variable "pn". We can now say that "pn" *points to* "n".

`*pn=6`

this puts the integer value "6" into the variable *pointed to* by "pn". Notice that this accomplishes the same thing as doing `n=6`

MORE VERY IMPORTANT STUFF!!!

The `*pn` in the second assignment is called a *de-reference*. This is the other use of the "*". If you use "pn" then you are accessing the address that is stored in the pointer variable "pn". If you use "*pn" you

are accessing the data stored in the address that is stored in "pn". Please read this explanation three times and then tap your heals together and say "there's no place like home". You have to understand this 100% if you are going to be able to use pointers effectively. When you read "*pn" read it as "at pn" and it will make sense. Here is how you should draw pictures to help you understand what we have done here so far:

I call this a "memory map". You should refer to this picture throughout this discussion to help your understanding. As the pointer applications get more complex, drawing memory maps will be a valuable skill to have. Notice also from the picture why we call them "pointers"!

Here is a printf( ) that will print the address of "n" twice and the value in "n" twice. Try reading it out loud using my hints:

```
printf("%d %d %d %d\n",pn,&n,*pn,n);
```

Even though addresses are technically not integers, you can print them with %d.

Now, write a short program with all the code we have used here and finish it off with the above printf( ). Regardless of how complex the application may be, if you get confused about pointers, come back to this page and read it again. It really is a simple concept. It is the complex applications that give pointers a bad name!

Note: Have you started looking at the C pointers tutorial by Ted Jansen? Check the "Start Here" section under "About Text Books" for the link.

File: pointers - malloc and arrays

pointers - malloc and arrays

Pointers - malloc( ) and arrays

dynamic memory allocation with `malloc( )`

The *malloc( )* function is for "**m**emory **alloc**ation. malloc( ) reserves memory space while the program is running. When you use "int" or "float" etc. to create a variable, this space is actually reserved during compilation. So, if we wanted to create an integer location while the program is running (hence the term "dynamic") we would do:

```
pn=malloc(4);
```

instead of the pn=&n that we used in the last example. Now we have a true "referenced" variable, because the 4 bytes of memory space that we just reservered does not have a name. The only way we can get to it is by de-referencing pn.

malloc( ) takes as input the number of bytes to be reservered, and returns the address of the memory that it just reserved. Usually you use the "sizeof( ) function to indicate the number of bytes to be reserved. The sizeof( ) function takes a type name and returns the number of bytes in one item of that type. sizeof(int) would be 4 on most machines. The sizeof( ) function is good for making code portable, but also for figuring the size of userdefined data types, such as struct and union types.

The malloc( ) function will return a pointer of the same type as the variable that is being assigned. But, because many programmers are paranoid about type-matching, as I am, you will often see a cast with malloc( ), "just in case". So, the above malloc( ) will be written like this:

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

The "int*" cast ensures that malloc returns the correct pointer type, and the "sizeof(int)" ensures that malloc allocates the correct number of bytes for an integer.

Pointers and Arrays

If you do:

```
int data[100];
```

You create 100 integer memory locations **at compile time**
If you do:

```
int* data;
data=(int*)malloc(sizeof(int)*100);
```

You create 100 integer memory locations **during runtime**
Other than when they are created, both "arrays" are the same. In either case you can use array notation to access items in the "array". It is that simple. There is one subtle difference, and that is that with the

pointer, you can assign a new value to it. This would be bad because the pointer contains the address of the beginning of the 100 memory locations. If you put a new value in the pointer, then there is no way to access the 100 memory locations you reserved. This is called a memory leak. If something like this is happening inside a loop, or in an application that gets called frequently (opening a window) then the results can be disasterous.

File: Pointer notation vs Array notation

Pointer notation vs Array notation

Pointer notation vs Array notation

Array notation

If you want to access (print in this example) items in an array, you could write the following loop:

```
for (i=0;i<MAX;i++)
  printf("%d\n",data[i]);
```

The "i" is an *offset*. The final address is computed by taking the index "i", multiplying it by the size of a data item (4 in this case) and then adding it to the *base address* that is stored in "data". So, when you add one to "i", you add whatever the size of one data item is to the final address, so adding one to the index makes the final address point to the next item in the list. This notation will work, regardless of how the array "data" was declared.

Pointer notation

If you want to access (print in this example) items in an array, you could write the following loop:

```
ptr=data;
for (i=0;i<MAX;i++){
  printf("%d\n",*ptr);
  ptr++;
}
```

Here "ptr" is an integer pointer (int* ptr). When you incrament a pointer with "++" the value in the pointer variable is changed, not by one, but by the size of the data item that it points to. So in this example, ptr++ actually adds 4 to ptr so that the next time through the loop the *ptr will get the next item in the array. Again, this notation will work, regardless of how the array "data" was declared.

Usually, pointer notation is used when you do **not** know the number of items to be processed. So you would use a *do-while* or *while* loop and check for a null pointer at the end of the list. NULL is a standard constant which means esentailly that the pointer value is zero. We will assign the NULL constant to pointers so that we can identify the end of the list. In summary, you will most often use pointer notation like this:

```
ptr=data;
while (ptr!=NULL){
  printf("%d\n",*ptr);
  ptr++;
}
```

File: pointers to structures

pointers to structures

pointers to structures

The ugly notation

If you have the following structure declaration:

```
struct TIME {
  int hours;
  int minutes;
  int seconds;
};
typedef struct TIME Time;
```

and then you declare a pointer to "Time":

```
Time* ptr;
```

Then accessing members of the structure is a little complicated. Since the address of the struct is in the pointer variable, to get to the struct variable, we must de-reference the pointer **first** and then use the "." to specify the member we want to access. :

```
(*ptr).minutes=23;
```

The parentheses are necessary because the "." has precedence over the "*". Without the parentheses we would be trying to access a member of the variable "ptr" and since "ptr" is not a structure, we would get an error.

The pretty notation

Fortunately, C has a much nicer way to access the member of a structure through a pointer:
```
ptr->minutes=23;
```
means exactly the same thing as the above "ugly" code. The really nifty thing about this notation is that it looks like a "pointer". I like to read it "ptr arrow minutes". With any pointer to a struct or union, you can use either notation. It is instructive to make sure you understand exactly what the "ugly" notation is doing. Also remember that the "pretty" notation is just a definition. It doesn't really make sense to try to take it apart to understand it as you should do with the "ugly" notation. The arrow notation is just there to make ugly code read easier.

File: Arrays of Structures

Arrays of Structures

Arrays of Structures

Let's use our Time struct for our array example:
```
struct TIME {
  int hours;
  int minutes;
  int seconds;
};
typedef struct TIME Time;
```
To make an array of these structures we use the same syntax as we would with any array:
```
Time times[MAX];
```
The tricky part is accessing the items in the array. Remember that the variable "times" is an array so the index goes after "times". Since "times[i]" is a structure (there are MAX of them in the array) then we would access a member by using the "." after the index.
```
times[i].seconds=51;
```
The mistake that students often make is putting the "." after "times", which is wrong, because "times" is not a structure, it is an array of structures (a pointer to a struct, in fact).

File: command line parsing (ex. code "parser.c")

command line parsing (ex. code "parser.c")

Command line parsing

The *Command line*

The command line is the stuff you type to get a program to execute. You are probably used to just typing the program name, or clicking on an icon. Here we will be giving the program information on the command line. Just as we did with the file name in the dynamic program.
Example:
```
%prog1 Sean 41 1
```
Here the program "prog1" will now run and can access the strings "Sean" and "41" and "1". These are called "command line arguments".

UNIX type command options

If I want a listing of my directory, I type:
```
%ls
```
If I want to see alot more information about each file, called a "long" listing, I type:
```
%ls -l
```
Here the "l" obviously stands for "long" and is called an option. We can also give the "ls" command a directory name, and it will show a listing of that directory, and not my current directory:
```
%ls -l Dir1
```
The string "Dir1" is called and *argument*. Now, I'm showing you all this so you will understand the double meaning of *argument* and that a string on the command line can be an *option* and an *argument* at the same time. Even though unix calls the "-l" string an option, as far as the command (or program) is concerned, it is an argument. So we call it an argument or an option depending on the context we are

refering to it in. (that is a yucky sentence!)

Now that I have your attention, suppose I want to give a name, age, and month (as an int) to a program. I want to enter this information as command line arguments. How does the program know which argument is which? How do I remember how I wrote the program? What if I'm almost always entering the same value for one of them, do I have to type it every time? To answer all these questions, we will make the program not care what order we enter the arguments by preceding each argument with a unix-type option (which will also be an argument to the program). The easy way to do this is with single letters. Then the program knows that the argument following the "-n" option is the name. We will also set the arguments to some "default" values before we start parsing the command line, so that if the user decides not to enter one of the values, the variable will have the desired default value already. So, all of the following will work fine:

```
%prog1 -n Sean -a 41 -m 1
```
the name would be "Sean", age would be 41, and month would be January
```
%prog1 -m 1 -n Sean -a 41
```
the name would be "Sean", age would be 41, and month would be January
```
%prog1 -n Sean
```
the name would be "Sean", age and month would be the default values
```
%prog1 -a 41
```
the age would be 41 and month and name would be the default values
```
%prog1
```
all three would be the default values

How the heck we going to code all that?!

I thought you would never ask! The following code is in Code directory on the ce machine called "parser.c". Go through it carefully, copy, compile and run it too, and ask if there is anything you do not understand. Notice the following:

- the program checks to make sure there are not too many arguments
- the default values are set first
- the program loops through all the arguments (two at a time, because they occur in pairs)
- each option is checked to make sure it starts with a "-"
- each argument is acted on with the switch( ) and the correct variable is set
- the program uses the exit( ) function to send a value to the OS based on how the program failed. If you were to implement this as a function, you would want to use a retrun( ) so that the error condition can be checked by the calling function, usually main( )

```
main(int argc, char** argv)
{
int age,month,maxargs,i;
char name[80];

/*(NOTE:  this may look overkill  but please notice how easy
        it will be to add another command line option, or to
        add another 50 options!  If there is something
        you don't undestand, please mail me questions.
        I added the "main" stuff above and the printf()
        at the end, just so you can compile and run the
        program to see how it works.
*/
/*
current command line arguments:

         -n name          User's first name ("Sean" is default)
         -m month         integer month (default is "1" for January)
         -a age           integer age (default is "39" :) )
*/

 maxargs = 7;

/* check for too many command line arguments */
 if (argc > maxargs) {
   printf("Usage:  %s [-n name -m month -a age ]\n",argv[0]);
   exit(1);
 }

/* set default values */
 strcpy(name,"Sean");
 month = 1;
```

```
 age = 39;

 for (i=1;i<argc;i+=2) {

/* check for valid flags (first char should be '-' for flags) */
   if (argv[i][0]!='-') {
     printf("bad option character...should be \"-\"\n");
     exit(2);
   }

/* check for valid flag value and take appropriate action */
   switch (argv[i][1]) {
      case 'n': strcpy(name,argv[i+1]);
                break;
      case 'm': month = atoi(argv[i+1]);
                break;
      case 'a': age = atoi(argv[i+1]);
                break;
       default: printf("bad option...should be \"n\" or \"m\" or \"a\"\n");
                exit(3);
                break;
   }
 }
printf("age = %d, month = %d, name = %s\n",age,month,name);
}
```

File: Assignment 3

Assignment 3

Assignment 3

Problem solving in "C"
Assignment 3
10 points
Due Tuesday, September 22nd, 2009

The Assignment:

Write a function which takes a structure type "Time" and returns an int which is the total number of seconds in the structure. Write another function that takes a pointer to a structure type "Time" and returns the total number of seconds in the structure pointed to. Mail me the functions so I can test them. Feel free to mail early, but make SURE they compile with no errors or warnings. Use the include file time.h that is in the directory Code. Get this file by loggin on and typing:

```
cp ~smcgowan/Cyber/PS/Code/time.h .
```

and don't forget the "." at the end of the command. It refers to your current directory.
Please remember that to include a file from the standard include directories you use the angle brackets:

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

but to include a file from some other directory you use quotes with the path to the file included. In this case, put a copy of time.h in your current directory and use

```
#include "time.h"
```

Put both functions in the same file and call the functions ftime( ) and pftime( ). Please remember to mail me just the functions, no main( ). Also, before you email me the file that contains both functions, make sure you compile it:

```
cc -c functs-file.c
```

so that you know there are no compiler errors or warnings.

Notes:

"total number of seconds" means 3600 time hours + 60 times minutes + seconds
The best thing to do here would be write your own driver to test these before you submit them.

Print **Save to File**