Printable View of: Week 2: Creating a menu-driven interface

Print | Save to File

File: Macros

Macros

Macros

What is a *macro*?

As you may have seen in other computer related applications, a *macro* is simply a single command or statement that replaces several commands or statements. Please note that a **function** is not a macro. In higher level programming languages, such as C, a macro is implemented at compile time. The macro name is replaced by the macro definition before the machine code is written. With a function, the code for the function is translated directly into machine language. If you could look at the machine code, you could identify the functions, but you could not tell if there were any macros used writing the program.

What do I use in C to create a *macro*?

When you compile a C program, the first thing that happens is the C pre-processor goes through the code once and acts on any **pre-processor directives**. You have used the `#include` and `#define` pre-processor directives in the past. To create a macro we will use the `#define` directive.

The #define directive tells the preprocessor to repace one string with another wherever it occurs in the source file *after* it reads the directive. Since the pre-processor makes only one pass through the code, replacements could not be made in the code that physicaly precedes the directive. An example of this would be the constant value for an array bound:

```
#define MAX 250
int data[MAX];
```

Remember that the pre-processor simply does a character replacement. In the example above, the characters "MAX" are replaced with the characters "250". It is that simple. No type checking, no nothing. If the replacement causes a syntax problem, then the compiler will catch it later. So, if you are having a problem with the "expansion" of your macro, check the documentation of your compiler and figure out how to make it "show the expanded macro" so that you can see what your macro replacement really looks like. On the ce machine you would compile as follows:

```
%cc -source_listing -show expansion filename.c
```

This would create the file "filename.lis" which would contain the macro expansions.

a sample *macro* with arguments

The #define may seem a little limited until you learn that you can pass arguments to it. Macros take arguments in a very intuitive way. The macro:

```
#define SQR(x) (x)*(x)
```

and the macro call:

```
a=SQR(b)
```

would cause the expansion:

```
a=(b)*(b)
```

Now there are actually two character substitutions being made.

1. Because the paramater "x" occurs in the macro **name**'s argument list, anything in the macro **call**'s argument list will replace any "x"'s in the macro **definition**. (Read that last sentence several times refering to the example until you have a clear understanding of all the terms.)
2. Then, the once-substituted macro definition replaces the macro call in the source code.

This brings us to the issue of the parenthesis. **All** of them are necessary. Suppose you wanted to use the "SQR(x)" macro to compute the square of an expression like **b + c**. Let's remove the parentheses and try the substitution.

```
#define SQR(x) x*x
```

the macro call:

```
a=SQR(b+c)
```

would cause the expansion:

```
a=b+c*b+c
```

You remember from your grammar school math that multiplications are done before additions, so this expression would not result in the squaring of the quantity (a+b). So, always remember to use parentheses around the macro argument **everywhere** that it occurs in the macro definition.

File: Example program "dynamic".c

Example program "dynamic"

Example program "dynamic"

Where is the *dynamic* program

All the code for the *dynamic* program is in the ~smcgowan/Cyber/PS/Code/Dynamic directory. This is a good time to mention that from now on when I refer to a directory on the ce machine, I will leave off the "~smcgowan/Cyber/PS part, because that is where everything you need will be and I'm getting tired of typing it. Now, as for the *dynamic* code, you can copy it all with the command:

```
%cp -r ~smcgowan/Cyber/PS/Code/Dynamic .
```

This will create a "Dynamic" directory in your directory. You can use

```
%cd Dynamic
```

to get into the directory and then you can use *cat* or *page* to view the source.

What does the *dynamic* program do?

The program will open a file (it gets the file name from the command line) and read an alphabetized list of names from the file. After reading the names it will

display the list on the screen
ask you to enter a name to be added to the list
display the list on the screen
ask you to enter a name to delete from the list
display the list on the screen
ask you to enter a name to be added to the list
display the list on the screen

This is all very nice, but not very useful if you want to do anything other than add two names and delete one. Also, the new list is never saved anywhere. I wonder how we could fix this? (wink, wink)

How does the *dynamic* program work

The short answer is, "it isn't important right now". The program uses a linked-list data structure to manage the list of names. Later on in the course we will lean to handle linked-lists, but for now we are just concerned with makeing *dynamic* more user friendly, and this task has nothing to do with the guts of how the data is handled. The files that comprise the program are:

dynamic.c
readlist.c
writelist.c
add.c
delete.c
dynamic.h

and the only one you will need to change is dynamic.c (you will need to write two new functions as well). Please remember this when you are looking over the code and working on assignment #1.

How do you create the *dynamic* executable?

To create the executable program *dynamic* you need to compile all the source together. We could do

```
%cc dynamic.c readlist.c writelist.c add.c delete.c -o dynamic
```

This would create the executable file called *dynamic* which would contain the machine code from all the C source in all the files listed. This is wonderful, except that every time you change one C source file you would have to compile all of them again. This isn't such a huge deal for five files, but imagine if we had 500. That would be a huge waste of cpu time.

One solution would be to create an object file for each source file. An object file is the compiled machine code that has not been "linked" or "loaded" together with all the other pieces of machine code. If you do:

```
%cc -c dynamic.c
```

(the -c option is for "compile only" ie supress the linking phase or in other words, do not create an executable file) you create a file called *dynamic.o* and you could do this for all the source files. Then you would do:

```
%cc dynamic.o readlist.o writelist.o add.o delete.o -o dynamic
```

Then, when you edit one of the files, you only need to create one new .o file and use the above cc command to "re-link" the new .o file with all the other .o files that haven't changed. This is better, but still takes lots of typing on your part, as well as remembering what files you changed since the last time you compiled.

The way to go is to use the unix program make which is the next topic in this lesson :)

File: Unix tools: make

Unix tools: make

Unix tools: make

NOTE: please read the following *lightly* because make is a really easy command to use but a little tricky to understand. I explain it here, but all you need to do is be able to use it, you do **NOT** need to understand it or be able to write your own makefiles.

What does *make* do?

The make program looks for a file called "makefile" or "Makefile" and reads that file and does what that file tells it to do. The key is that make checks the time stamps on files so that it knows what source files need to be re-compiled and then it links all the objects (new and old) together to make (hence the name) the new executable. So, it does the least amount of work necessary and all you need to type is four characters! *Wahoooo!*

How does *make* work?

The make program reads directions from the file "makefile" by default. These directions are in the following form:

```
target-file:dependency list
rule
```

If you are working with C programs, the target file would be the object file. The dependency list would contain the C source file and any user-defined header (.h) files that may be included in that file. make checks the time stamps on all the files in the dependency list. If any of these files are *newer* than the target file (the object file) then the *rule* is executed. In our case, the rule will be a cc command that will re-create the target object file. For example:

```
readlist.o:readlist.c dynamic.h
cc -c readlist.c
```

So, if either dynamic.h or readlist.c is changed, and you type *make*, then the command *cc -c readlist.c* is executed and you get a new *readlist.o*.

If you look at the file *makefile* in the Code/Dynamic directory you see that the first line has *dynamic* as the target file and **all** the object files in the dependency list. So, if any of the objects have been updated (because make found a source file that was updated) then the rule that "links" all the objects together to create a new *dynamic* will be executed.

Using *make* for the dynamic program

Once you have copied the Code/Dynamic directory to your directory, cd to Dynamic and simply type

```
%make
```

The make program will read the file "makefile" and compile all the source files and create the executable file called dynamic which you can then execute as follows:

```
%dynamic list.dat
```

"list.dat" is just a sample file I included for testing. You can make your own, more interesting file and use it by putting it's file name on the command line. Try running the program without typing a file name and notice the error message you get. I tried to make this look like a typical unix "usage" error message. Aren't I special? Actualy, this is one of those things you do to make things run more "seemlessly".

You will notice how limited the functionality of this program is, but it does work properly. You will also notice that there is a file called "makefile.new" which we will use once we upgrade the dynamic program by creating two new functions. The makfile.new file contains the dependency list and rule for these 2 functions. You can either use the unix command:

```
%mv makefile.new makefile
```

to rename the file mafefile.new to "makefile" so that when you type *make* the make program will read the new makefile. Or, you can tell make to use a file other than the default by typing the command:

```
%make -f makefile.new
```

This is what you will need to do to get your upgraded version of *dynamic* compiled.

File: Unix tools: mailx

Unix tools: mailx

Unix tools: mailx

Notes:

There many unix mail programs available. Please notice that `"Mail"` and `"mailx"` are the same program and vastly different from "mail". "mail" has an unfriendly interface and limited functionality. Make sure you are typing `"Mail"` or `"mailx"`.
You do not need to master all this today. :) Just read through this and use this page as a reference. This will all be second nature by the end of the course.

Unix mail tutorial

1. To enter the mailx utility you simply type "mailx" at the prompt. If you have no mail messages, the system will tell you that and give you the system prompt back.

```
%mailx
no mail for username
%
```

2. To send a message to someone just type

```
%mailx username
Subject:some important stuff
Dear Someone,
blah
blah
blah
Me
^d
%
```

notice that the mailx program prompts you for a subject. To finish and send the message, you type control-d

3. After you get mail from someone, if you start mailx, the mailx program will show you a list of your messages.
EXAMPLE:

```
%mailx
Mail version SMI 4.0 Thu Oct 11 12:59:09 PDT 1990  Type ? for help.
"/usr/spool/mail/smcgowan": 3 messages 3 unread
>U  1 bpantano           Wed Sep  9 14:00   52/1553  assn1
 U  2 bstuhl             Fri Sep 11 10:04   13/293   test
 U  3 bpantano           Fri Sep 11 10:04   13/299   ru
? q
Held  3 messages in /usr/spool/mail/smcgowan
%
```

Notice that the messages are numbered, the subject and the sender are displayed as well. You can read a message by typing the number of the message at the ? prompt. You can get this listing again by typing h at the ? prompt. If you type ? at the ? prompt you get a list of commands that you can use.

4. The current message is the one that the > is pointing at. You can reply to the current message by typing an r or R (r replys to the entire list when the message you get was sent to a bunch of users, R replys to just the sender) Use R to send a message back to one of the people that sent you a message.

5. You leave mailx by typing q for quit or an x for exit. The x will not save any changes to your mail file (ie a message you deleted is still there)
   If you read a message but don't delete it, it is saved to a file called mbox when you leave mailx.
   You can read messages in mbox by typing
   ```
   %mailx -f mbox
   ```
   the -f option tells mailx to open a file that is not your system mail box file.
   Read a message and do not delete it and exit mailx with a q. Now type *mailx -f mbox* and see that the message it there. Now, delete it and mailx will tell you that mbox has been removed because there are no more messages in it.

6. You save a message to a file by using an s followed by the name of the file you want to put it in. You can also save with a w, but this does not save the message header. When you use the s or w, the current message is saved. Remember that the current message is the one that has the > pointing to it. Now have someone send you another message and you start mailx and save it to a file. If the filename that you give to mailx is the name of a file that already exists, the saved message will be APPENDED to the end of the file. If the file does not exist, mailx will create it.

7. To mail a file you use the unix input redirect character <
   ```
   %mailx -s "subject here" smcgowan < file.name
   ```
   You will use this command to submit assignments. Please remember to include a good subject. For an internet based course, this is of utmost importance. If you are submitting an assignment, use "PS: assn# submit". If you are sending me email that is not an assignment submission, and you need a response, please use the word **question** or **help** in the subject, so that I know you need a quick response.

File: Unix tools: pico text editor

Unix tools: pico text editor

Unix tools: pico text editor

Why use *pico*?

The *pico* editor is very user friendly. You aren't taking a unix course, so there is no real reason to kill yourself trying to learn to use a powerful tool such as *vi* that takes forever to learn. Save that for your unix class, or for your abundant spare time. :)

How do you use *pico*?

To edit an existing file or to create a new one you type:
```
%pico filename.c
```
You will see the contents of the file on the screen and a list of commands across the bottom of the screen. There isn't much more to it. Again, it isn't a very powerful editor, but it doesn't take any time to learn.

Suppose I were silly enough to try to use *vi* anyway?

If you do have time to spend learning *vi* I have a nice tutorial you can get with:
```
%cp ~smcgowan/Cyber/Unix/vitutor .
```
Once you copy the *vitutor* file, you just type:
```
%vi vitutor
```
and follow the instructions **CAREFULLY**! Allow yourself about 45 minutes for your first run through the tutorial. Do the tutorial several times. The more you do it, the better you will get at the editor. Good luck!

File: Assignment 1

Assignment 1

Assignment 1

Problem solving in "C"
Assignment 1
10 points

Due Tuesday, September 15th, 2009

The Assignment:

Write functions menu( ) and writefile( ) for program dynamic.c. You should not change ANYTHING in any of the existing functions, except cleaning up the main body. In main( ), you need to remove everything after the call to readlist( )and replace what you removed with a call to menu( ). For the output you should write the list to the screen, do at least one add and one delete (not the same name), write the list to the screen and then write the list to the file. Hand in the source, output, and the new copy of list.dat.
Have fun, mail me questions.

Descriptions of the new functions

menu( )

This function should take a pointer to the list and the name of the file containing the original list (this is exactly like the readlist( ) function). menu( ) needs the pointer to the list so it can send that pointer to the other functions. menu( ) needs to know the name of the file so that it can send the filename to writefile( ) when the user wants to "save" the current list.

The menu( ) function needs to display a list of choices for the user:

add a name
delete a name
show the list
save the list
quit

then get the user's input, then call the appropriate function, and then repeat the process until the user chooses the "quit" option. Do this by putting a switch statement inside a do-while loop. Use the original main( ) function as a guide for how to call all the other functions. The function declaration should look like this:

```
void menu(Person*,char*)
```

writefile( )

This function is almost identical to the writelist( ) function. The only changes are that you need to declare a file pointer (type `FILE*`) and use fopen( ) to open the file

```
fp=fopen(filename,"w");
```

and use `fprintf( )` to write to the file (instead of printf( ) to write to the screen).

```
fprintf(fp,"control string",arg1,arg2,etc);
```

Do not forget to take out the printf's that identify the output, because everything you write to the file will be interpreted as a name for the list the next time you run the program. Also, remember to `fclose( )` the file when you finish writing the list. The function declaration should look like this:

```
void writefile(Person*,char*)
```

Notes:

Include only the appropriate function declarations in main( ) and menu( )
Please ask questions before you spend too much time on any one problem
If you have trouble with writing to a file in the `writefile()` function, please check the `readlist()` function. The implementation is all there. The only difference is that readlist( ) reads from the file and writelist( ) writes to the file.
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 the file dynamic.h should be in your current directory and use

```
#include "dynamic.h"
```

Check the notes on "make" so that you can quickly compile your new *dynamic* program

File: Assignment 2

Assignment 2

Assignment 2

Problem solving in "C"
Assignment 2
5 points
Due Tuesday, September 15th, 2009

The Assignment:

Write a function to compute the absolute value of an integer value. Then write a macro to compute the absolute value of an integer value. Write a driver to test them both. The function declaration should look like this:

```
int abs_val(int)
```

Notes:

- Do not use any standard functions such as abs( )
- The function and macro should be completly separate. i.e. do not write a function that simply invokes a macro.
- the best way to implement the macro is with a conditional operator:
  `expression?statement:statement`
- A "driver" is just a main( ) function that calls other functions, in this case just to test them. So, make sure your driver does a good job of testing the function and macro.

Chat Room: Week 2 Chat

chatroom

Print **Save to File**