CS 2110 Lab 15
Intro to C

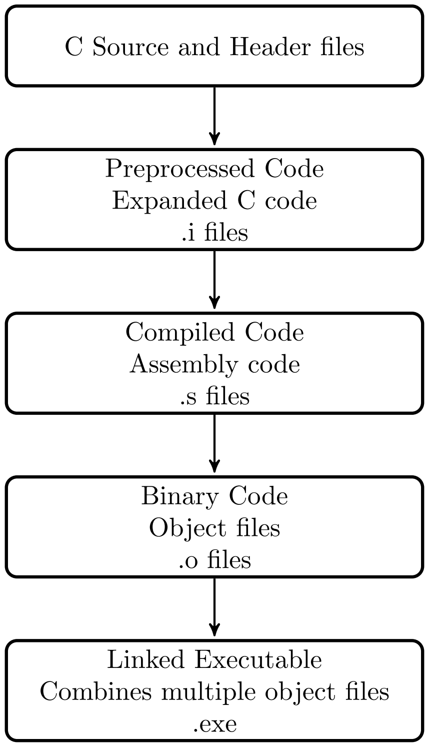
The TAs :)

10/03/2018

1 Recitation Outline

1. C Compilation
2. Memory Regions in C
3. Types in C
4. Macros
5. Structs
6. Typedef

2 C Compilation



2.1 C Source and Header files

2.1.1 C Source Files

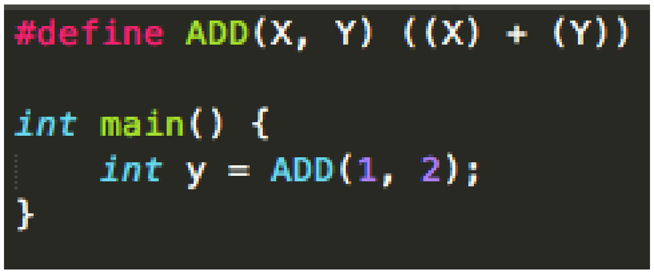
C source files are your typical files where all of your functions will be written.

2.1.2 C Header Files

C header files contain function declarations and global variables to be shared between several source files. In order to use a header file, you need to include it with `#include` at the top of your source file. For system header files, you include them via the `#include <file>` form, while the form `#include "file"` is used to include header files of your own program.

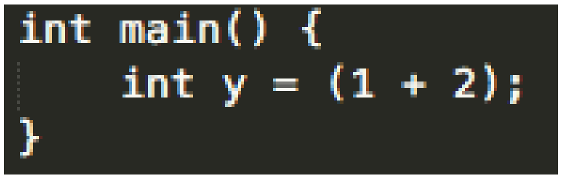
2.2 Preproccessed Code

Preprocessed code will be the result after condition checks have been executed (#if/#endif), files have been included (#include), and macros have been substituted.



```c
#define ADD(X, Y) ((X) + (Y))

int main() {
    int y = ADD(1, 2);
}
```



```c
int main() {
    int y = (1 + 2);
}
```

The above example shows a .c file before and after being preprocessed. Notice the macro has been replaced.

2.3 Compiled Code

Compiled code will be our .c code translated into the lovely assembly we're all familiar with! This assembly won't be LC3 assembly, however, since most modern computers don't use the LC3 datapath.

2.4 Binary Code

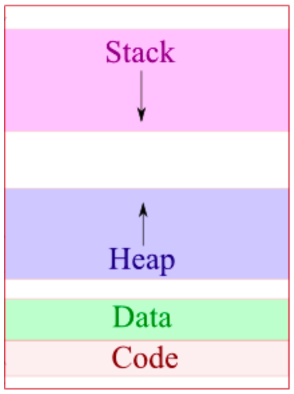
Binary code is the assembly code from compilation, disassembled into its binary translation. This is the same process you did when you translated LC3 assembly instructions into 16 bits.

2.5 Linked Executable

The binary code is then made into an object file. However, object files themselves are not executable. The Linker takes multiple object files (say for example you had multiple .c files that worked together) and creates one executable file to run your program!

You can see for yourself these intermediate files by using the flag: -S -save-temps when compiling your files with gcc!

3 Memory Regions in C



Memory in C is broken up into 4 different sections.

1. **Stack**: The stack is what we used with the LC3 calling convention! This is where local variables to a function will be stored. It's important to note that variables stored here will only last the lifetime of a function call (Remember tearing down the stack!), so never reference a location in memory on the stack outside the lifetime of a function call.
2. **Heap**: The heap is where dynamically allocated memory will be. When large chunks of memory too big for the stack or chunks of memory that need to live longer than the lifetime of a function call are needed, they live in the heap. This is where our handy-dandy friend malloc will get memory from!
3. **Data**: Data is where global variables and static variables will be stored. Similar to the heap, variables stored here will outlive function calls. So a static variable, for example, will persist across multiple calls to the same function.
4. **Code**: The last region, code, is simply where your executable code gets stored!

4 Types in C

4.1 Static

4.1.1 Static defined on a function

Static functions are not visible outside of the C file they are defined in. (Think similar to private in this use case)

4.1.2 Static defined inside a function on a local variable

Statically defined local variables do not lose their value between function calls, since they are not stored on the stack!
Example:

```
int function(void) {
    static int x = 0;
    x++;
}
```

Every time function is called, x will be incremented by one.

```
function(); // x = 1
function(); // x = 2
function(); // x = 3
function(); // x = 4
```

4.1.3 Static defined outside a function

Static global variables are not visible outside of the C file they are defined in. (Think similar to private in this use case)

4.2 volatile

Tells the compiler that the value may change at anytime, so it should not try to optimize the value away.

4.3 extern

The extern modifier tells the compiler that the variable is declared in another file.

4.4 auto

Local variables that are always stored on the stack. All variables are "automatically" declared auto unless they are declared something else.

4.5 const

This simply defines a variable as constant, meaning it cannot be modified.

5 Macros

Macros are a preprocessor directive. Essentially that means they tell the preprocessor to do something *before* compilation. Specifically, macros can be thought of as advanced text replacement. You define macros the following way:

```
#define MACRO_NAME(ARGUMENTS) TEXT_REPLACEMENT
```

The key thing to remember is that macros are essentially just text replacement, and are processed before compilation. Therefore to prevent unintended behaviors from order of operations, you should **always** surround the entire expression and every use of an argument in parentheses. For example, say you didn't follow these rules and were making a multiply macro:

```
#define MULTIPLY(X, Y) (X*Y)
.
.
.
int sum = MULTIPLY(3+4, 5+6);
```

this code would get preprocessed into:

```
int sum = 3+4*5+6;
```

which would be 3 + 20 + 6 = 29, instead of 7 * 11 = 77.

6 Structs

A data type in C that can contain multiple variables of different types. These variables are stored in one block in memory that is the struct. You can access these variables using the '.' operator, similar to accessing variables of a class in Java! It's also important to know that the name of the corresponding type includes struct as well as the struct name. An example of a struct would be:

```
struct myStruct {
    int number;
    char* letter;
};
```

Then you could create a variable of struct myStruct and set its number like:

```
struct myStruct myVar;
myVar.number = 5;
```

7 Typedef

A keyword that allows you to create a new type that is an alias for an existing one. Example:

1.
   ```
   typedef unsigned char BYTE;
   ```

   This allows the identifier BYTE to represent the type unsigned char

2.
   ```
   typedef struct Dog {
       int tails;
       int feet;
   } Puppy
   ```

   With this typedef, you can use the identifier puppy to represent the type struct Dog.