

Makefiles, and .h files, and
.c files, and .o files, OH MY!

For projects with more complexity.
(Great.. Just what we needed)

Breaking your program into files

- main.c
- stack.c
- stack.h

Breaking your program into files

- I have an example on
~cs157/class/7_Makefile
 - main.c
 - The main function, to actually do the “job”
 - stack.c
 - The code for a stack of integers.
 - stack.h
 - The “declarations” of a stack of integers.

Why break them up?

- main just needs a stack
 - It does not want (or need) to care how it is built or used!
- Smaller files are easier to read
- Faster to compile
 - More on this later
- Breaks the program into logical CHUNKS

stack.h

```
typedef struct S_stack {  
    int number;  
    struct S_stack *next;  
} stack;
```

```
void push(int number, stack **stk_ptr);  
int pop(stack **stk_ptr);
```

- No actual code!
- Just “this is the structure” and
- These are the functions. ... “never mind how they work”

stack.c

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

#include "stack.h"
```

- Why include stack.h?
- Note the “” instead of <>
 - <> means “include from the system libraries”
 - For predefined .h files
 - “” means “include from THIS directory”
 - For your OWN .h files

stack.c

```
void push(int number, stack **stk_ptr) {
    stack *stk, *tmp;
    stk = *stk_ptr;
    tmp = malloc(sizeof(stack));
    tmp->number = number;
    tmp->next = stk;
    stk = tmp;
    *stk_ptr = stk;
}
```

stack.c

```
int pop(stack **stk_ptr) {  
    int number;  
    stack *stk, *tmp;  
    stk = *stk_ptr;  
    tmp = stk;  
    number = tmp->number;  
    stk = stk->next;  
    free(tmp);  
    *stk_ptr = stk;  
    return number;  
}
```


main.c

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

#include "stack.h"
```

- Why include stack.h this time?

main.c

```
int main() {
    stack *stk = NULL;
    push(7, &stk);
    push(2, &stk);
    push(9, &stk);
    push(12, &stk);
    printf("%d\n", pop(&stk));
    printf("%d\n", pop(&stk));
    printf("%d\n", pop(&stk));
    printf("%d\n", pop(&stk));
    printf("%d\n", pop(&stk));
    return 0;
}
```

Compiling multiple files (Opt 1)

- `gcc -Wall main.c stack.c`
 - Compiles BOTH files... and makes a.out
- Advantages:
 - Easy to remember
- Disadvantages:
 - If you have a LOT of .c files, then it becomes tedious AND slow!

Compiling multiple files (Opt 2)

- `gcc -Wall -c main.c`
 - turns `main.c` into `main.o`
- `gcc -Wall -c stack.c`
 - turns `stack.c` into `stack.o`
- `gcc -Wall -o stacktest stack.o main.o`
 - takes `stack.o` and `main.o` and makes “`stacktest`” out of them
 - Called “LINKING”

Whats a .o?

- An “Object File”
- Contains the compiled contents of the corresponding .c program
- For example:
 - stack.o contains the computer-language version of stack.c
- Can't turn a .h into a .o (no code in .h)

Compiling multiple files (Opt 2)

- Advantages:
 - Faster (Only recompile parts then re-link)
- Disadvantages:
 - Loads of typing!

Makefiles

- Automate the process
- You tell the Makefile:
 - What you want to make
 - How it goes about making it
- And it figures out
 - What needs to be (re) compiled and linked
 - What order to do it in
- You just type “make”

Makefiles

- Can be HUGELY complex
- Just use the one I give you, and only modify the top parts
- Makefiles could be a class on their own...

Makefile

```
CC          = gcc
CFLAGS     = -Wall
LDFLAGS    =
OBJFILES   = stack.o main.o
TARGET     = stacktest

all: $(TARGET)

$(TARGET): $(OBJFILES)
    $(CC) $(CFLAGS) -o $(TARGET) $(OBJFILES) $(LDFLAGS)

clean:
    rm -f $(OBJFILES) $(TARGET) *~
```

Makefile

```
CC      = gcc ←
CFLAGS  = -Wall
LDFLAGS =
OBJFILES = stack.o main.o
TARGET  = stacktest
```

Which compiler to use

```
all: $(TARGET)
```

```
$(TARGET): $(OBJFILES)
```

```
$(CC) $(CFLAGS) -o $(TARGET) $(OBJFILES) $(LDFLAGS)
```

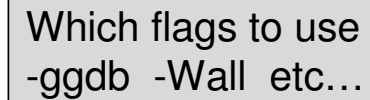
```
clean:
```

```
rm -f $(OBJFILES) $(TARGET) *~
```

Makefile

```
CC      = gcc
CFLAGS  = -Wall
LDFLAGS =
OBJFILES = stack.o main.o
TARGET  = stacktest
```

Which flags to use
-ggdb -Wall etc...



```
all: $(TARGET)
```

```
$(TARGET): $(OBJFILES)
```

```
$(CC) $(CFLAGS) -o $(TARGET) $(OBJFILES) $(LDFLAGS)
```

```
clean:
```

```
rm -f $(OBJFILES) $(TARGET) *~
```

Makefile

```
CC      = gcc
CFLAGS  = -Wall
LD_FLAGS = ←
OBJFILES = stack.o main.o
TARGET  = stacktest
```

Which libraries to use
-lm -lelfence etc...

```
all: $(TARGET)
```

```
$(TARGET): $(OBJFILES)
```

```
$(CC) $(CFLAGS) -o $(TARGET) $(OBJFILES) $(LD_FLAGS)
```

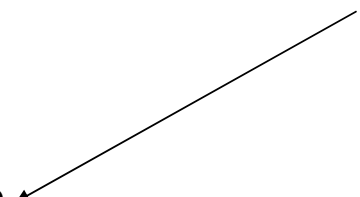
```
clean:
```

```
rm -f $(OBJFILES) $(TARGET) *~
```

Makefile

```
CC      = gcc
CFLAGS  = -Wall
LDFLAGS =
OBJFILES = stack.o main.o
TARGET  = stacktest
```

Which object files are
part of the final program



```
all: $(TARGET)
```

```
$(TARGET): $(OBJFILES)
```

```
$(CC) $(CFLAGS) -o $(TARGET) $(OBJFILES) $(LDFLAGS)
```

```
clean:
```

```
rm -f $(OBJFILES) $(TARGET) *~
```

Makefile

```
CC      = gcc
CFLAGS  = -Wall
LDFLAGS =
OBJFILES = stack.o main.o
TARGET  = stacktest
```

What to name
the final prog

```
all: $(TARGET)
```

```
$(TARGET): $(OBJFILES)
```

```
$(CC) $(CFLAGS) -o $(TARGET) $(OBJFILES) $(LDFLAGS)
```

```
clean:
```

```
rm -f $(OBJFILES) $(TARGET) *~
```

Makefile

```
CC      = gcc
CFLAGS  = -Wall
LDFLAGS =
OBJFILES = stack.o main.o
TARGET  = stacktest

all: $(TARGET)

$(TARGET): $(OBJFILES)
^ $(CC) $(CFLAGS) -o $(TARGET) $(OBJFILES) $(LDFLAGS)

clean:
^ rm -f $(OBJFILES) $(TARGET) *~
```

TAB
not several spaces
Sorry...

To use our Makefile:

- Just type “make”
 - It will figure out which .c files need to be recompiled and turned into .o files
 - If the .c file is newer than the .o file or
 - the .o file does not exist
 - Figures out if the program needs to be re-linked
 - If any of the .o files changed or
 - If the program does not exist

To use our Makefile:

- Or type “make clean”
 - Deletes:
 - all the .o files
 - all the ~ files (from emacs)
 - the program itself
 - Leaves:
 - .c files
 - .h files
 - Makefile

To use our Makefile:

- make clean
- make
- What happens?