Today's Goals

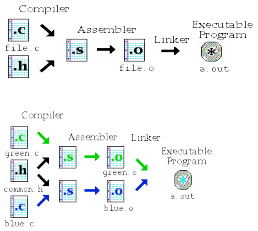
- Multiple source files
  - Splitting your code
  - Header files
  - Sharing information
  - Makefile
- Writing large programs
  - Modules

Section 1

The Compilation Process

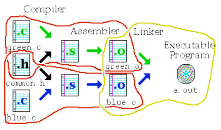
- Compiler:
  - All **`.c`** files are converted/assembled into Assembly Language, i.e. making **`.s`** files.
- Assembler:
  - The assembly language files from the previous step are converted into object code (machine code), i.e. **`.o`** files.
- Linker:
  - The object code is then linked to libraries and other files for cross-reference.

Compilation



Compiler/Assembler and Linker

- Compile green.o: **`cc -c green.c`**
- Compile blue.o: **`cc -c blue.c`**
- Link together: **`cc green.o blue.o`**



Section 2

Multiple Source Files

- The decision to divide your program into multiple source files is not only a matter of size.
- One and only one **`.c`** file must contain a **`main`**.
- Functions that handle some common aspects of a program should be grouped into the same file.
  - main, data structure implementation (i.e. linked list), I/O, utilities, display/GUI, etc

Header Files

- To share information between files.
  - types
  - macros
  - functions
  - externals
- Each **`.c`** should have its own **`.h`**.
- Information share btw. several or all files should go into one **`.h`** (usually **`main.h`**).

Types and Macros

- Types:
  - **`typedef`**
  - **`enum`**
- Macros
  - **`#include`**
  - **`#define`**
  - **`#ifdef`**
  - **`#error`**

Sharing Functions

- If a function is to be called in more than one file, put its prototype into a **`.h`**.
- Always include the **`.h`** with **`f`**'s prototype in the **`.c`** that contains **`f`**'s definition.
  - For any **`.c`**, always include your own **`.h`**.
- A header file should never contain function definitions.

Sharing Variables

- Variables shared between files are defined in one file, and declared in all files that need to access it.
  - Definition of a variable causes the compiler to set memory aside
- **`extern`**
  - **`extern int i, a[];`**
  - extern informs the compiler that the variables **`i`** and **`a`** are defined elsewhere.

`extern` variables

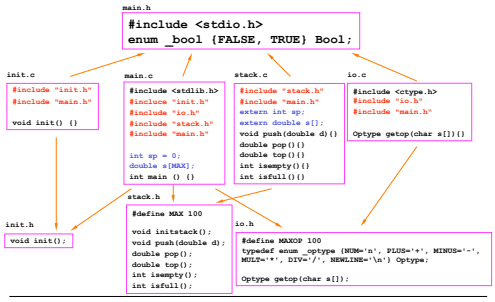
- **`extern`** declarations often go in to a header file.
- The variable must have one (and only one) definition among all files.
  - **`int x;`**
- Any file that wishes to access a variable that is defined in another file must declare such a variable as **`extern`**
  - **`extern int x;`**

Section 3

Example

- The implementation of a stack-based calculator:
  - **`1 2 - 4 5 + * ==> (1-2) * (4+5)`**
- Two globals:
  - **`double s[MAX];`**
  - **`int sp = 0;`**
- Stack related operations
- I/O operations

Program Structure



main.c

```
int main() {
  Optype t; char str[MAXOP]; double d;
  init();
  while((t = getop(str)) != EOF) {
      switch(t) {
      case NUM:
        push(atof(str)); break;
      case PLUS:
        d=pop(); push(pop()+d); break;
      case MINUS:
        push(pop()-pop()); break;
      case NEWLINE:
        printf("\t%.2f\n", pop()); break;
      default:
        fprintf(stderr, "Error, unknown command %s\n", str); break;
      }
  }
  return 0;
}
```


io.c

```
Optype getop(char s[]) {
  int i=0; char c;
  while ((s[0] = c = getchar()) == ' ' || c == '\t') ;
  s[1] = '\0';
  if (!isdigit(c) && c != '.') /* not a number */
    return c;
  if (isdigit(c))  /* collect int part */
    while(isdigit(s[++i] = c = getchar())) ;
  if (c == '.')    /* collect fractional part */
     while(isdigit(s[++i] = c = getchar())) ;
  s[i] = '\0';
  if (c != EOF)
    ungetc(c, stdin);
  return NUM;
}
```


Protecting your header files

- Always enclose your **.h** with these directives:

```
#ifndef NAME_H
#define NAME_H
/* header file contents */
#endif
```

- **#error –** to check for conditions under which the header file shouldn't be included

```
#ifndef DOS
#error Graphics supported only under DOS
#endif
```


Section 4

Building a Multiple-File Program

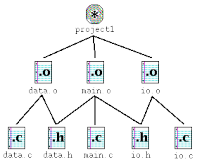
- Makefile
  - List all source files to be compiled and linked
  - Lists dependencies among all files

```
calc: main.o init.o io.o stack.o
        cc –o calc main.o init.o io.o
  stack.o
main.o: main.h init.h io.h stack.h
        cc –c main.c
```

  - target: list of files
  - build/rebuild command

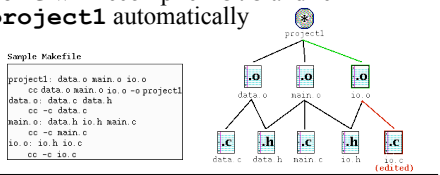
Dependency Graph

- The principle by which Make operates
- In writing a Makefile, you are specifying the dependencies needed to build your executable



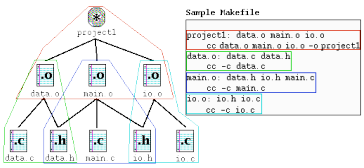
Updates According to Dependencies

- Suppose you edited **io.c**
- **Make** realizes the update based on timestamp of **io.c**
- **Make** will recompile **io.o** and relink **project1** automatically



Dependencies in **Make** syntax

- **target: source file(s)**
  command (tabs in front!!)



Makefile Flags/Macros

- **CC = gcc**
- **CFLAGS = -g -Wall -ansi**
- **-D** – allows the value of a named macro to be specified
  - **-DDEBUG=1 == -DDEBUG**
- **-UD** – undefines a named macro
- **$(CC) $(CFLAGS) –DDEBUG –c main.c**

Section 5

Programming-in-the-large

- Most full-featured programs today are at least 100,000 lines long.
- Careful design
  - Do not limit any possible future extensions
- Extreme attention to style
  - Lots of people are going to be working on it
- Planning for maintenance
  - It IS going to be modified many times

Module-based Programming

- A module is a collection of services (functions), some of which are available to other parts of the program (clients) through an interface (header file).
  - Abstraction – carefully designed modules should hide implementations
  - Reusability – in order to maximize this feature, modules should be as basic and as minimal as possible
  - Maintainability – easier to locate and fix bugs

Cohesion and Coupling

- High cohesion
  - Elements in each module should be closely related – (i.e. all the stack functions)
- Low coupling
  - Modules should be as independent of each other as possible

Summary

- Learn to master the structure of a multiple-file program.
- Properly setting up the “bones” of a program will extend the initial set-up time, but one ends up saving time and frustration in the end.
- Think 3 times before you start, and if you have partners, meet 5 times before you code.