# `#define`

- Often used to define constants
  - `#define TRUE 1 #define FALSE 0`
  - `#define PI 3.14159`
  - `#define SIZE 20`
- Offers easy one-touch change of scale/size
- `#define` vs constants
  - The preprocessor directive uses no memory
  - `#define` may not be local

## `#define` makes it more readable

```
#include<stdio.h>
#define MILE 1
#define KM   2

void km_mile_conv(int choice) {
   // …
   if (choice == MILE)
   // …
}
int main() {
   // …
   switch (choice) {
   case MILE:
      km_mile_conv(choice);
      break;
   caea KM:
      km_mile_conv(choice);
      break;
   /* more cases */
   }
}
```

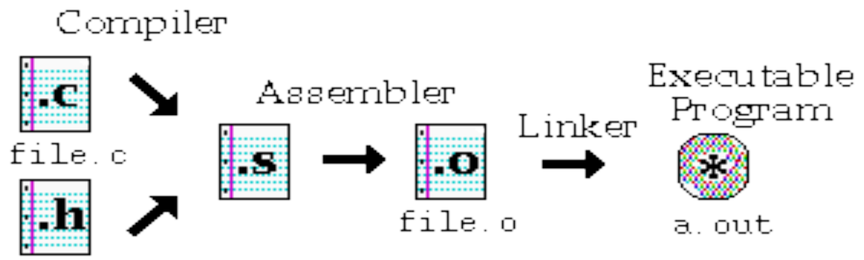
## Program Organization

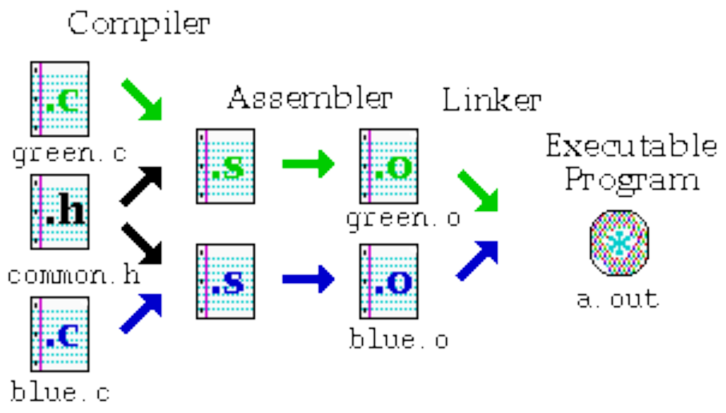
- `#include` and `#define` first
- Globals if any
- Function prototypes, unless included with header file already
- `int main()` – putting your `main` before all other functions makes it easier to read
- The rest of your function definitions

## 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.

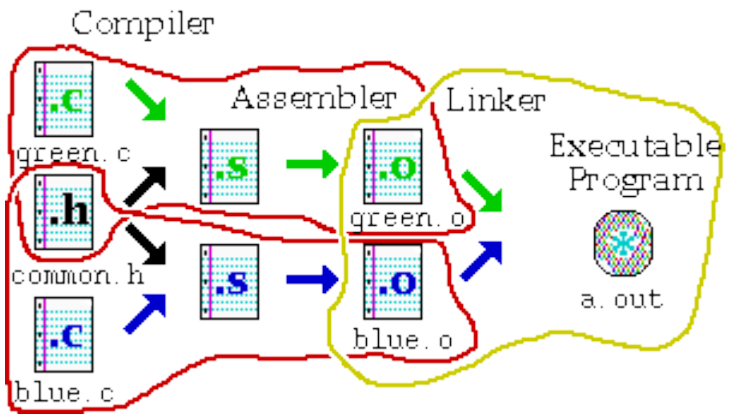
# Compilation





## 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`**



## 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 calls **`f`**.
  - 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.

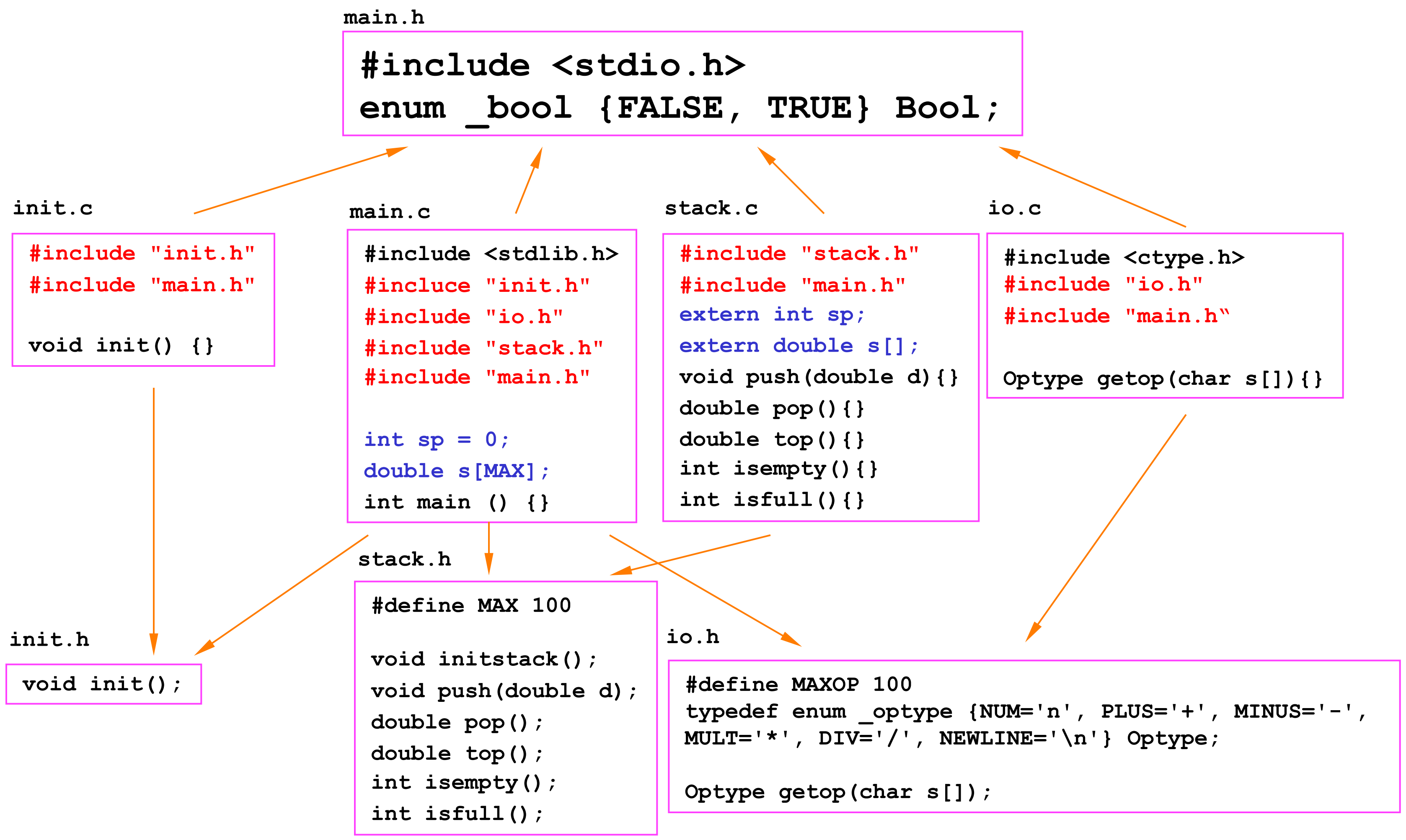
## `extern` variables

- **`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;`**

## 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.h
```
#include <stdio.h>
enum _bool {FALSE, TRUE} Bool;
```

init.c
```
#include "init.h"
#include "main.h"

void init() {}
```

main.c
```
#include <stdlib.h>
#incluce "init.h"
#include "io.h"
#include "stack.h"
#include "main.h"

int sp = 0;
double s[MAX];
int main () {}
```

stack.c
```
#include "stack.h"
#include "main.h"
extern int sp;
extern double s[];
void push(double d){}
double pop(){}
double top(){}
int isempty(){}
int isfull(){}
```

io.c
```
#include <ctype.h>
#include "io.h"
#include "main.h"

Optype getop(char s[]){}
```

init.h
```
void init();
```

stack.h
```
#define MAX 100

void initstack();
void push(double d);
double pop();
double top();
int isempty();
int isfull();
```

io.h
```
#define MAXOP 100
typedef enum _optype {NUM='n', PLUS='+', MINUS='-',
MULT='*', DIV='/', NEWLINE='\n'} Optype;

Optype getop(char s[]);
```

## 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
```

## Building a Multiple-File Program

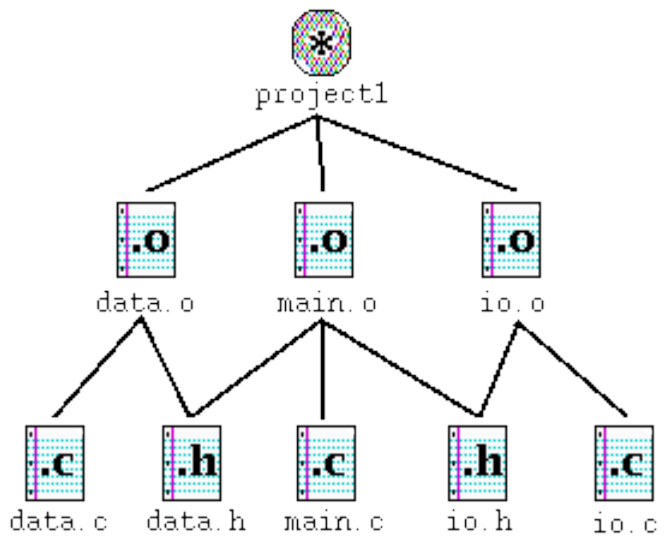
- Makefile
  - List all source files to be compiled and linked
  - Lists dependencies among all files

```
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

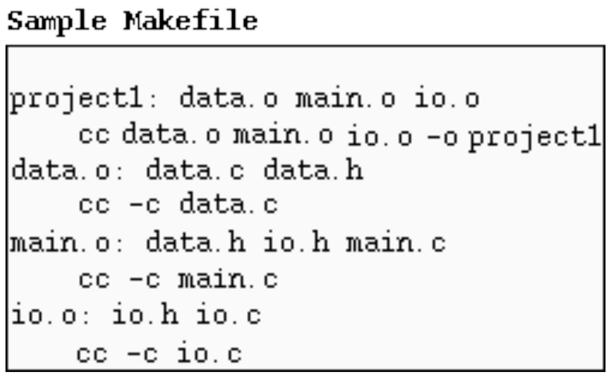
## Dependency Graph

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



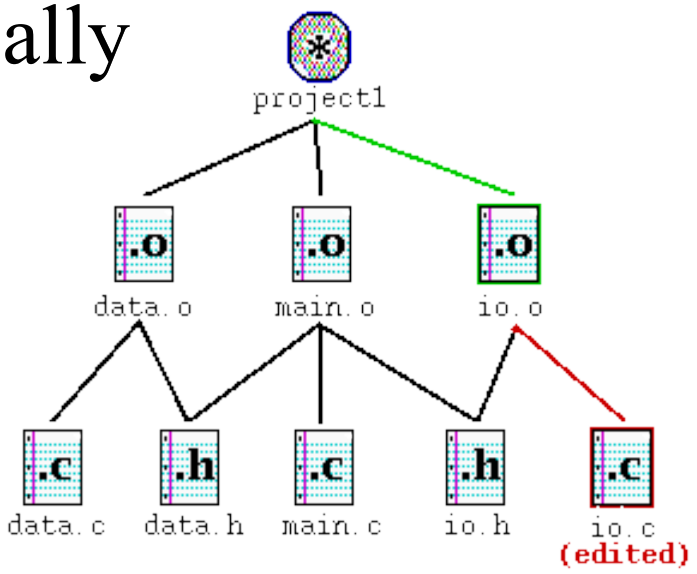
# Updates According to Dependencies

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



**Sample Makefile**

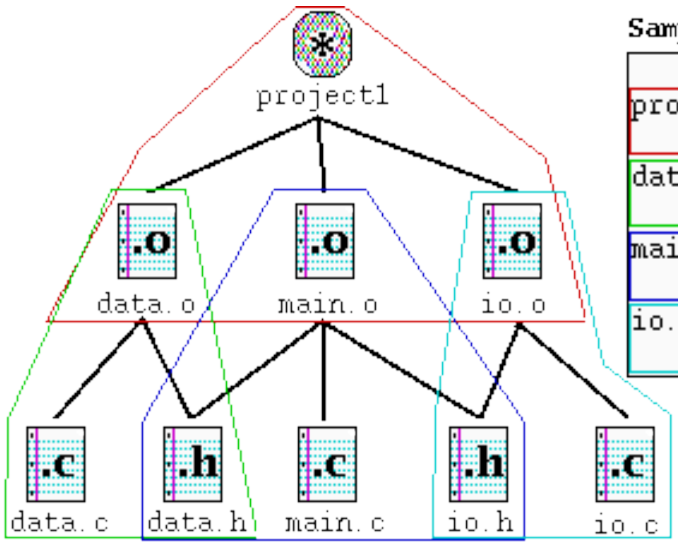
```
project1: data.o main.o io.o
     cc data.o main.o io.o -o project1
data.o: data.c data.h
     cc -c data.c
main.o: data.h io.h main.c
     cc -c main.c
io.o: io.h io.c
     cc -c io.c
```

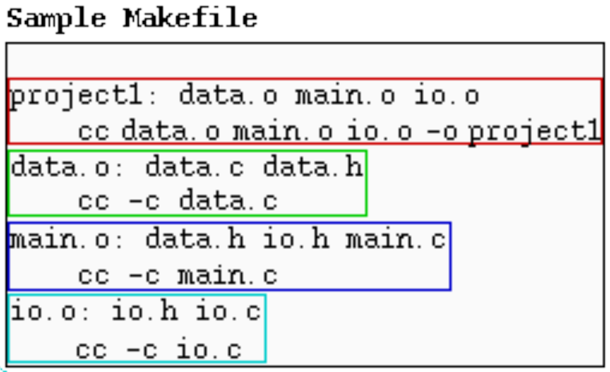


# Dependencies in **Make** syntax

- **`target:     source file(s)`**

  **`command (tabs in front!!)`**





Sample Makefile

```
project1: data.o main.o io.o
        cc data.o main.o io.o -o project1
data.o: data.c data.h
        cc -c data.c
main.o: data.h io.h main.c
        cc -c main.c
io.o: io.h io.c
        cc -c io.c
```

## 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`**

## #define

```
#define ESCAPE_KEY 27
#define FILLED 1

enum{FALSE,TRUE}
enum{BLACK,RED,BLUE,GREEN};
Enum{LINE,TRIANGLE,RECTANGLE};
```

## Globals

```
int id = 0;                                // debug id
int fillmode = FILLED;
int color = BLACK;
int mode = LINE;
Shape *shapes = NULL;
V2d *vs = NULL;

char *c[]={"black","red","blue","green"};
char *m[]={"line","triangle","rectangle"};
GLfloat glc[][3]={{0.0,0.0,0.0},{1.0,0.0,0.0},
                  {0.0,0.0,1.0},{0.0,1.0,0.0}};
```

## Write Functions to Draw Primitives

```
void draw_point(V2d v) {
  glBegin(GL_POINTS);
  glVertex2i(v.x, v.y);
  glEnd();
}
void draw_line(Ln *line, int fmode) {
  if (!fmode)
    glEnable(GL_LINE_STIPPLE);
  glBegin(GL_LINES);
  glVertex2i(line->v1.x, line->v1.y);
  glVertex2i(line->v2.x, line->v2.y);
  glEnd();
  glDisable(GL_LINE_STIPPLE);
}
```

## Write Functions to Draw Primitives

```
void draw_triangle(Trig *triangle, int fmode) {
  if (fmode)
    glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
  else
    glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);

  glBegin(GL_POLYGON);
  glVertex2i(triangle->v1.x, triangle->v1.y);
  glVertex2i(triangle->v2.x, triangle->v2.y);
  glVertex2i(triangle->v3.x, triangle->v3.y);
  glEnd();
}
```

# Linked List Loop

```
glColor3fv(glc[s->color]);

switch(s->type) {
case LINE:
  draw_line(((Ln *) s->shape), s->fillmode);
  break;
case TRIANGLE:
  draw_triangle(((Trig *) s->shape), s->fillmode);
  break;
case RECTANGLE:
  draw_rectangle(((Rect *) s->shape), s->fillmode);
  break;
default:
  break;
}
```