

# Week 6: Miscellaneous C and Makefiles

- Assignment 2 is posted
- There is a lab for this week. It will be posted later today.
  - short PCRS question on Makefiles
  - complete some code to submit on MarkUs (related to A2)

# External and static variables

- **External variable**: declared outside the body of a function
- **File scope**: visible from the point of the declaration to the end of the file.
- **Static storage duration**: through the duration of the program.
- External/global variables have file scope and static storage duration.

# static variables

```
static int i;
```

```
void f(void) {  
    static int j;  
}
```

- static used outside a block means that the variable is only visible in the file in which it is declared
- static used in a block means that the variable lives beyond the duration of the block, and is initialized only once.

# Example

```
#include <stdio.h>

int nextvalue()
{
    static int i = 0;
    i++;
    return i;
}

int main()
{
    int i;
    for(i = 10; i > 0; i--) {
        printf("%d\n", nextvalue());
    }
    return 0;
}
```

output:

1  
2  
3  
4  
5  
6  
7  
8  
9  
10

# extern

filea.c

```
extern int i;  
  
void f(void) {  
    i++;  
}
```

fileb.c

```
int i = 0;  
extern void f(void);  
  
void g(void) {  
    f();  
    printf("%d\n", i);  
}
```

- informs the compiler that `i` is an `int` variable, but doesn't cause it to allocate space.

# typedef

- You can define new types using typedef.
- You have already seen the effects of typedef  
`typedef unsigned int size_t;`

- Example

```
struct personrec {  
    char name[20];  
    int age;  
};  
typedef struct personrec Person;  
Person *p = malloc(sizeof(Person));
```

# typedef

- Example

```
typedef struct {  
    char name[20];  
    int age;  
} Person;
```

```
Person *p = malloc(sizeof(Person));
```

# Header files

- When you begin to split up your C program into multiple files, you need header files to store function and type declarations.

main.c

```
void add(int);  
int isEmpty();  
extern List *head;  
int main()  
{  
    add(10);  
    isEmpty();  
    head = NULL;  
}
```

list.c

```
List *head = NULL;  
int isEmpty()  
{...}  
void add(int v)  
{...}  
void remove(int v)  
{...}
```



## list.h

```
struct node {
    int value;
    struct node * next;
} ;
typedef struct node List;
extern List *head;
int isEmpty(int);
void add(int);
void remove(int)
```

## main.c

```
#include "list.h"

int main()
{
    add(10);
    isEmpty();
    head = NULL;
}
```

## list.c

```
#include "list.h"
List *head = NULL;
int isEmpty()
{...}
void add(int v)
{...}
void remove(int v)
{...}
```

list.h

```
struct node {
    int value;
    struct node * next;
} ;
typedef struct node List;
List *head = NULL;
int isEmpty();
void add(int);
void remove(int)
```

Wrong!

main.c

```
#include "list.h"

int main()
{
    add(10);
    isEmpty();
}
```

list.c

```
#include "list.h"

int isEmpty()
{...}
void add(int v)
{...}
void remove(int v)
{...}
```

Wrong!

main.c

```
#include "list.c"

int main()
{
    add(10);
    isEmpty();
}
```

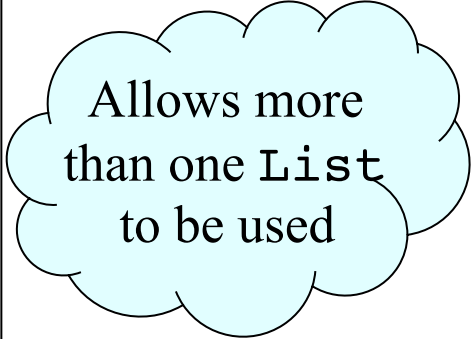
list.c

```
List *head = NULL;
int isEmpty()
{...}
void add(int v)
{...}
void remove(int v)
{...}
```

## list.h

```
struct node {
    int value;
    struct node * next;
} ;
typedef struct node List;

int isEmpty(List *);
void add(List *, int);
void remove(List *, int)
```



Allows more  
than one List  
to be used

## main.c

```
#include "list.h"

int main()
{
    List *list1 = NULL;
    add(list1, 10);
    isEmpty(list1);
}
```

## list.c

```
#include "list.h"

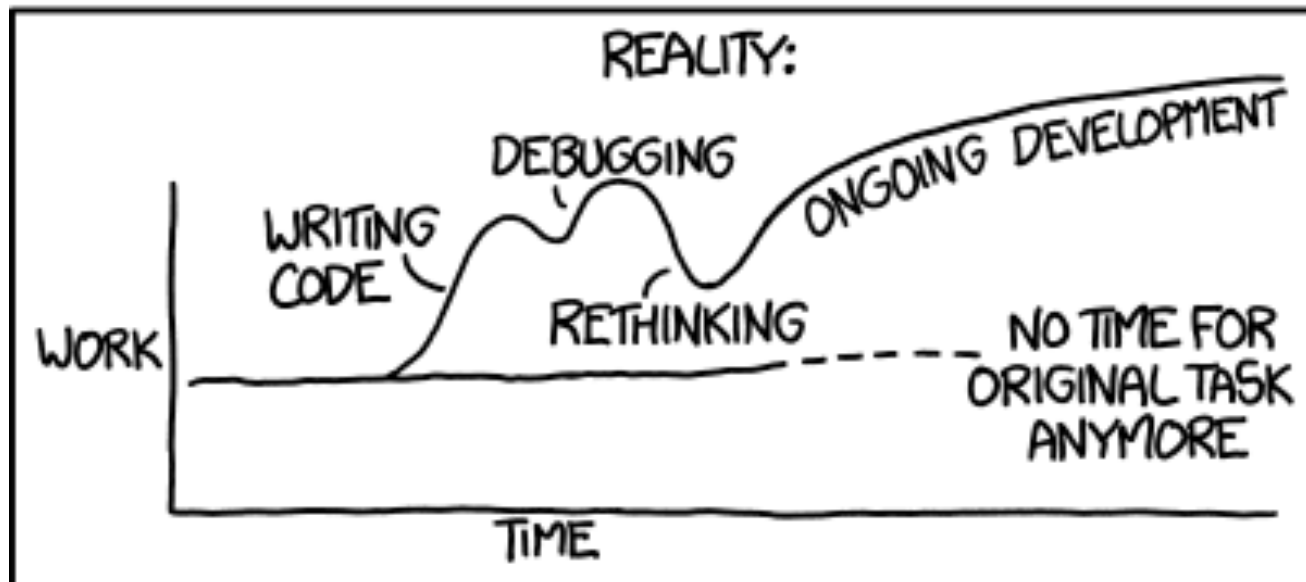
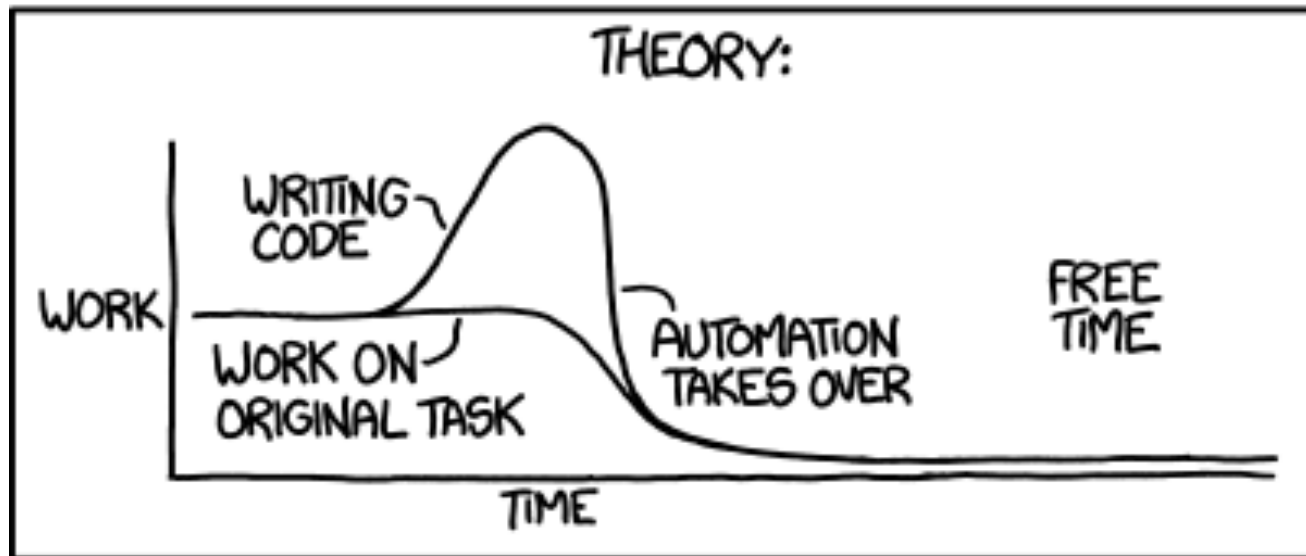
int isEmpty(List *h)
{...}
void add(List *h, int v)
{...}
void remove(List *h, int v)
{...}
```

# Protecting header files

- Compilation errors may result if a header file is included more than once.
- This causes a problem if the header file defines types.
- Use preprocessor directives to selectively compile.

```
#ifndef LIST_H  
#define LIST_H  
...(contents of the header file)  
#endif
```

"I SPEND A LOT OF TIME ON THIS TASK.  
I SHOULD WRITE A PROGRAM AUTOMATING IT!"



# Makefiles

- Makefiles were originally designed to support separate compilation of C files.

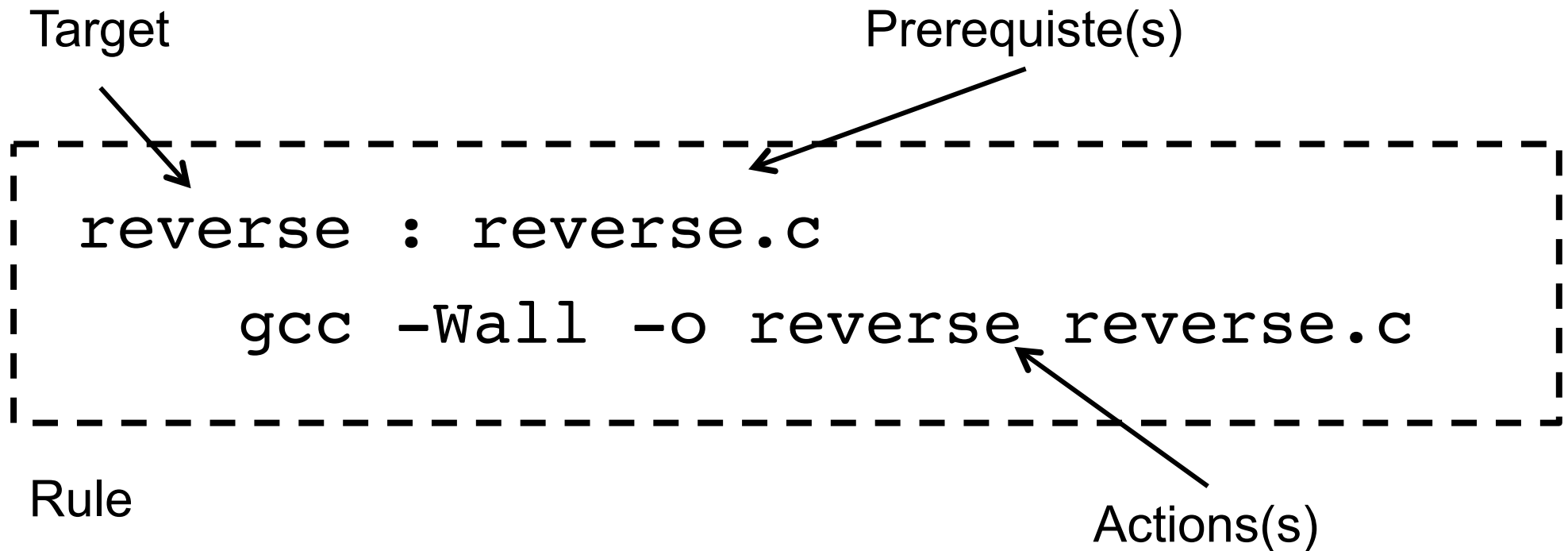
```
testlist : testlist.o list.o test1.o
    gcc -Wall -g -o testlist testlist.o list.o test1.o
```

```
testlist.o : testlist.c
    gcc -Wall -g -c testlist.c
```

```
list.o : list.c list.h
    gcc -Wall -g -c list.c
```

```
test1.o : test1.c list.h
    gcc -Wall -g -c test1.c
```

# Terminology



- May be many prerequisites (dependency)
- Rule may have many actions (one per line)



# Running make

- `make`
  - with no options looks for a file called `Makefile`, and evaluates the first rule
- `make query`
  - Looks for a file called `Makefile` and looks for a rule with the target `query` and evaluates it.

# How it works

- Make looks at the when the target and its prerequisites were last modified
  - It assumes targets are files and checks the dates of the files
- Make does nothing...
  - If the target exists, and
  - Is more recent than all its prerequisites
- Make executes the actions...
  - If the target doesn't exist, or
  - If any prerequisite is more recent than the target

# Variables

```
CFLAGS= -g -Wall
```

```
reverse : reverse.c
```

```
gcc ${CFLAGS} -o reverse  
reverse.c
```

Make defines variables to represent parts of rules

\$@	Target
\$<	First prerequisite
\$?	All out of date prerequisites
^	All prerequisites

# Example

all : packetize read\_stream

packetize : packetize.o crc16.o list.o data.h

gcc -Wall -g -o packetize packetize.o list.o  
crc16.o

read\_stream : read\_stream.o crc16.o list.o data.h

gcc -Wall -g -o read\_stream read\_stream.o  
list.o crc16.o

crc16.o : crc16.c data.h

gcc -Wall -g -c crc16.c

list.o : list.c data.h

gcc -Wall -g -c list.c

packetize.o : packetize.c data.h

gcc -Wall -g -c packetize.c

read\_stream.o : read\_stream.c data.h

gcc -Wall -g -c read\_stream.c

# Example

```
all : packetize read_stream
```

```
packetize : packetize.o crc16.o list.o data.h
```

```
gcc -Wall -g -o $@ $^
```

```
read_stream : read_stream.o crc16.o list.o data.h
```

```
gcc -Wall -g -o $@ $^
```

```
%.o : %.c data.h
```

```
gcc -Wall -g -c $<
```

# Pattern rules

Most files are compiled the same way

So write a pattern rule for the general case

```
% .o : % .c
```

```
gcc ${CFLAGS} -c $<
```

Use % to mark the stem of the file's name

Like using \* in commands in Unix

-c flag in gcc does compilation of file without linking.

# Multiple Targets and Phony Targets

- Often you want one command to build a number of other targets

```
all : packetize readstream
packetize : ...
...
readstream : ...
```

Or targets aren't building anything

```
clean:
    rm *.o packetize readstream
```