Header Files and Make Utility

Header files

- Organize code
- Provide function prototypes
- Shared declarations
- Promoting modularity
- Code reuse

Makefile

- Rules to compile parts of a program
- Issues command to recompile
- Automate software building procedure
- Automate complex tasks
- Contain
  - Dependency rules
  - Macros
  - Suffix or implicit rules

Header Files

- Essential for modular programming which enhances code organization, readability, and maintainability
- Provides a way to reuse code and ensures consistency
- Declare functions, variables, and settings that can be shared across multiple source files
- "Instructs” compiler on how to interface with library and user-written components
- Programmer-defined header files (.h)
  - Custom header files which allow personalization
  - Included by using `#include "filename.h"` directive

Here's a basic structure of a header file:

```
// example.h
#ifndef EXAMPLE_H
#define EXAMPLE_H

// Declarations
void myFunction(int x);

// Constants
#define MAX_VALUE 100

#endif // EXAMPLE_H
```

Explanation:

- `#ifndef` , `#define` , and `#endif` are used for header guards, preventing multiple inclusions
- Inside the header file function prototypes and constants allow other source files to use them

In your C source file:

```c
// main.c
#include <stdio.h>
#include "example.h"

int main() {
    myFunction(MAX_VALUE);
    return 0;
}
```

Explanation:

- `#include` is used for standard or custom header files
- Use functions and constants declared in the header file in your source file

Simple Program to Deconstruct

```c
#include <stdio.h>

void myPrintHello(void);

int main() {
  myPrintHello();
}

void myPrintHello(void) {
  printf("Hello MakeFiles!\n");
}
```

Deconstruction

```c
/* hellomain.c */
#include "helloheader.h"

int main() {
   myPrintHelloMake();
   return(0);
}

/* hellomyprint.c */
#include <stdio.h>
#include "helloheader.h"

void myPrintHelloMake(void) {
  printf("Hello MakeFiles!\n");
}

/* helloheader.h */
void myPrintHelloMake(void);
```

Complexity of the Program

- Modularized programs have plenty of benefits
- Testing can become more difficult
- Compile commands need to be exactly the same
- `-I` option added to gcc command
  - include directory for header files

```
gcc -o hello hello.c
./hello
rm hello

gcc -o hello hellomain.c hellomyprint.c helloheader.h
./hello
rm hello

gcc -o hello hellomain.c hellomyprint.c helloheader.h -I
./hello
rm hello
```

Makefiles

- Makefile is a special type of script
- Can be used with may different languages
- Write custom makefiles to compile programs
- File must be `makefile` or `Makefile`
- Command `make` triggers the script
- Ensures efficient and organized compilation
- Manages dependencies and automates the build process
- Saves time by only recompiling necessary files

Makefile Examples

```
target... : prerequisites
        recipe
```

```
hello: hello.c
        gcc -o hello hello.c -I.

.PHONY: clean

clean:
        rm -f hello
```

```
hellos: hellomain.c
        gcc -o hello hellomain.c hellomyprint.c -I.

.PHONY: clean

clean:
        rm -f hellos
```

Variables in Makefiles

- Define variables for compiler, flags, etc.
- Increases flexibility and maintainability
- Example using variables in a Makefile

```
CC=gcc
CFLAGS=-I.

hellos: hellomain.o hellomyprint.o
	$(CC) -o hellos hellomain.o hellomyprint.o $(CFLAGS)

hellomain.o: helloheader.h hellomain.c
	$(CC) -c hellomain.c $(CFLAGS)

hellomyprint.o : hellomyprint.c
	$(CC) -c hellomyprint.c

.PHONY: clean

clean:
	rm -f *.o hellos
```

```
CC=gcc
CFLAGS=-I.
DEPS = helloheader.h
OBJ = hellomain.o hellofunc.o

hellos: $(OBJ)
        $(CC) -o hellos $(OBJ) $(CFLAGS)

hellomain.o: $(DEPS) hellomain.c
        $(CC) -c hellomain.c $(CFLAGS)

hellomyprint.o : hellomyprint.c
        $(CC) -c hellomyprint.c

.PHONY: clean

clean:
        rm -f hellos $(OBJECTS)
```

```
CC=gcc
CFLAGS=-I.
OBJ = hellomain.o hellomyprint.o
PROG = hellos

$(PROG): $(OBJ)
        $(CC) -o $(PROG) $(OBJ) $(CFLAGS)

hellomain.o: helloheader.h hellomain.c
        $(CC) -c hellomain.c $(CFLAGS)

hellomyprint.o : hellomyprint.c
        $(CC) -c hellomyprint.c

.PHONY: clean

clean:
        rm -f $(PROG) $(OBJ)
```

```
CC=gcc
CFLAGS=-I.
DEPS = helloheader.h
OBJ = hellomain.o hellomyprint.o

%.o: %.c $(DEPS)
	$(CC) -c -o $@ $< $(CFLAGS)

hello: $(OBJ)
	gcc -o $@ $^ $(CFLAGS)

.PHONY: clean

clean:
	rm -f *.o hellos
```

```
CC = gcc
CFLAGS = -Wall
PROG = program1 program2 program3

all: $(PROG)

program1: program1.o
	$(CC) $(CFLAGS) -o $@ $^

program2: program2.o
	$(CC) $(CFLAGS) -o $@ $^

program3: program3.o
	$(CC) $(CFLAGS) -o $@ $^

.PHONY: clean

clean:
	rm -f $(PROG)
```

Explanation

- `CC` is the compiler variable (gcc in this case)
- `CFLAGS` contains common flags (-Wall for warnings)
- `PROG` lists the names of your C programs
- `all` target builds all programs specified in `PROG`
- Individual targets for each program with specific source files are defined.
- `$@` and `$^` are automatic variables representing the target and dependencies
- You can customize this Makefile based on your project structure and specific compiler flags for each program.

Questions

GNU Make Manual

http://www.gnu.org/software/make/manual/make.html