

Independent compilation

- Large programs are difficult to maintain
- Problem solved by breaking the program into separate files
- Different functions placed in different files
- The `main` function appears in only one file, conventionally known as `main.c`
- Advantages
 - Reduction in complexity of organizing the program
 - Reduction in time required for compilation
 - * Each file is compiled separately
 - * Only those files are recompiled that have been modified
- Compiler creates object files corresponding to each source file
- The object files are linked together to create the final executable
- Compilation time is reduced because linking is much faster than compilation
- Source files are compiled separately under Unix using the `-c` option to the compiler

```
gcc -c main.c
```

- The entire sequence of commands to create an executable can be specified as

```
gcc -c main.c  
gcc -c func.c  
gcc -o prog main.o func.o
```

Header files

- Used to keep information common to multiple source files
- Files need the same `#define` declarations and the same type declarations (`struct` and `typedef`)
- More convenient to declare these declarations in a single header file
- Header file can be used in all the files using `#include`
- Such a process avoids duplication and allows for easier modification since a constant or type declaration need only be changed in one place
- Guidelines for good header file usage
 - Header files should contain only
 - * Constant definitions
 - * Type declarations
 - * Macro definitions
 - * Extern declarations
 - * Function prototypes
 - Header files should not contain
 - * Any executable code

- No function definitions
- * Definition of variables
 - Only exception is to declare variables
 - Every variable declaration should be an `extern` declaration
 - Inclusion of variable definitions in header file causes multiple definitions of the same symbol which is a linkage error
- Organization of header files
 - More a matter of style
 - Preferable to have a logical organization
 - By convention, the files have a suffix `.h` but it is not required by the C preprocessor
 - * It is also recommended as some utilities (such as `make`) distinguish between C source files and header files using this convention
 - Advisable to split the header file into multiple header files for large projects
 - * `const.h` – for constant definitions
 - * `types.h` – for type definitions
 - * `extern.h` – for external variable declarations
 - Common to define all global variables in the file `main.c`
 - Preferable order of inclusion
 - * Include files in the following order


```
#include <stdio.h>
#include "const.h"
#include "types.h"
#include "extern.h"
```
 - * Important because types may need constants, and `extern` declarations may need types
- Preprocessor trickery
 - Alternative to defining all global variables in `main.c`
 - Use a preprocessor trick to cause `extern.h` to both declare and define global variables
 - The file of `extern` declarations has entries like


```
extern int x;
```
 - The variables are not defined in `main.c` or any other file; instead the lines of code shown below are placed in `main.c` (or the source file containing `main()`)


```
#define extern      /* Define extern to nothing */
#include "extern.h"
#undef extern       /* Revert to no change for safety */
```
 - * The first line defines `extern` to nothing or whitespace
 - * This has the effect of deleting all occurrences of the word `extern` in the header file
 - * Any `extern` declaration without the keyword `extern` is a definition
 - * In all files except `main.c`, the variable `x` is qualified by `extern`, and the global variables are defined exactly once
 - Disadvantages of this technique
 - * Global variables cannot be easily initialized at compile time
 - * Initializations can be included with more preprocessor trickery but may not be worth the trouble

```
extern int x          /* no semicolon    */
#ifdef extern
    = 2              /* initialize    */
#endif
;                    /* end of declaration */
```

* This is the template to declare each variable

- Header files of function prototypes
 - Function prototypes need to be included to allow proper type checking
 - Prototypes are strongly recommended to remove the problem of [accidentally] using a function before it is defined
 - Omission of function prototypes loses all type checking of function arguments and may cause compiler or run-time errors
 - It is strongly recommended to maintain a header file containing a prototype for every function
 - No strict need to include prototypes in the files where the functions are defined but this is useful in checking that the declarations in the header file match the actual definitions
- Automatic generation of header files
 - Possible by using the `grep` and `sed` utilities to extract all function definitions
 - All you need to do is to extract the function definitions and add a semicolon at the end
 - Assumptions
 - * Function definitions start at the first character of a line
 - * The entire list of function parameters are on a single line

The Make Utility

- Utility to aid in the design and maintenance of multiple-file programs
- Relieves the programmer of the burden of remembering which files need recompilation
 - Examines the date of modification of files to see if they have changed
 - Recompiles the files if above is true
- Requires the use of a special file (called description file) in the directory, called `Makefile` or `makefile`
- Examines `Makefile` to determine the dependencies needed to build the executables
 - Sorts out dependency relations among files
 - Automatically recompiles only those files that have been changed after building the executable, and links them into the executable
- Based on non-procedural programming techniques, such as backtracking used in logic programming languages

The Makefile

- Used to make a *target* of the operation, using *prerequisites* or *dependents*; the dependents may have other files as prerequisites
- Sensitive to dependencies such as

```
source  ⇒  object
object  ⇒  executable
```

- Consists of two main types of lines described in the following general syntax

```
target :  dep1 dep2 dep3 ...  # Dependency line
        cmd                    # Command line
```

1. Dependency lines

- Also known as *rule lines*
- Show the dependencies between files
- Exactly one line long; use of backslash as the last character is allowed to extend the line
- The dependencies must be satisfied, possibly by building them as targets in other entries, to build the current target
- A single target may appear on multiple dependency lines; however, only one of them can be associated with commands

2. Command lines

- Commands to be executed to compile the files
- Must be on a single line but the use of backslash as the last character of the line allows extended lines
- Must begin with a tab as the first character on line¹
- tabs in the Makefile can be checked by the command


```
cat -v -t -e Makefile
```

 - * -v and -e options cause all tabs to appear as ^I
 - * -e option places a dollar sign at the end of each line to enable you to see any terminating whitespace
- Multiple commands can be specified on successive lines to be executed in sequence
- Multiple commands can be specified on same line if they are separated by a semicolon from each other

- Comments can be specified by the # sign and extend to the end of line

- Blank lines

- Allowed in some places
- Should not separate a dependency line from its commands
- Should not separate lists of commands

- Targets without prerequisites

- Must have a colon
- Need not be filenames and if so, are always executed
 - * make treats every non-existent target as out-of-date target

- A simple Makefile

```
CC      = gcc
CFLAGS  = -g
TARGET  = math
OBJS    = main.o square.o cube.o

$(TARGET): $(OBJS)
    $(CC) -o $(TARGET) $(OBJS)

main.o: main.c
    $(CC) $(CFLAGS) -c main.c
```

¹Most important syntax rule of make

```

square.o: square.c
    $(CC) $(CFLAGS) -c square.c

cube.o: cube.c
    $(CC) $(CFLAGS) -c cube.c

clean:
    /bin/rm -f *.o $(TARGET)

```

- Dependency checking (using above Makefile as an example)
 - First target to be built is `math`
 - `make` checks to see if `math` exists in the current directory
 - If it exists, `make` checks `main.o`, `square.o`, and `cube.o` to see whether any of them are newer than `math`
 - The process in the above step is repeated for each of the `*.o` files as targets, checking their dependencies, using later entries in Makefile
 - Only after all the prerequisites have been verified and brought up-to-date, `make` will exit making sure that the file `math` is current
 - Order of checking dependencies is important, and chain of commands must be issued in the correct order
 - It should also be apparent by now why the target name (to the left of the colon) is the same as the filename resulting from the execution of a command
- Minimizing rebuilds
 - If some object files are used in multiple programs, they need to be compiled only once
- Invoking `make`
 - Any target in the Makefile can be built by using the command


```
make target
```
 - Several targets can be specified in a single invocation of `make`

```
make square.o cube.o
```
 - If target is not specified, `make` attempts to make the first available target in the Makefile

Macros

- Description file entries of the form


```
NAME = text_string
```
- Can be referred to subsequently by `$(NAME)` or `${NAME}`
 - Parentheses or braces can be dispensed with for macros with single letter names
 - Preferable to use braces as it allows the use of parentheses exclusively for library modules
- Permit variation from one build to the next, for example, changing the `DEBUG_FLAG` from `-g` to `-O`
- Can be used anywhere in the Makefile, on both the dependency lines as well as the command lines
- Conceptually, the `make` utility expands macros before any other processing
- Conventionally, macro names are typed in all upper case letters

- A pound sign end the definition and starts a comment
- Macro definition can be continued on the next line by using backslash as the last character on the current line
- A macro definition with no string after the equal sign is assigned the null string
- Order of definition of macros is not important
 - Cannot redefine a macro once it has been defined in the same file
 - Macro must be defined before any dependency line in which it appears

Suffix rules

```
CC = gcc
CFLAGS = -g
TARGET = math
OBJS = main.o square.o cube.o
.SUFFIXES: .c .o
```

```
$(TARGET): $(OBJS)
$(CC) -o $@ $(OBJS)
```

```
.c.o:
$(CC) $(CFLAGS) -c $<
```

```
clean:
/bin/rm -f *.o $(TARGET)
```

Library archive

```
CC = gcc
CFLAGS = -g
TARGET = math
OBJS = main.o
LIBOBJS = square.o cube.o
LIBS = -lmymath
MYLIBS = libmymath.a
LIBPATH = .
.SUFFIXES: .c .o
```

```
$(TARGET): $(OBJS) $(MYLIBS)
$(CC) -o $@ -L. $(OBJS) $(LIBS)
```

```
$(MYLIBS): $(LIBOBJS)
ar -rs $@ $(LIBOBJS)
```

```
.c.o:
$(CC) $(CFLAGS) -c $<
```

```
clean:
/bin/rm -f *.o $(TARGET)
```

Creating shared objects

```
CC = gcc
CFLAGS = -g
TARGET = math
OBJS = main.o
LIBOBJS = square.o cube.o
LIB = mymath.so
.SUFFIXES: .c .o

$(TARGET): $(OBJS) $(LIB)
$(CC) -o $@ $(OBJS) $(LIB)

$(LIB): $(LIBOBJS)
$(CC) -G -o $(LIB) $(LIBOBJS)

square.o: square.c
$(CC) -fpic -c square.c

cube.o: cube.c
$(CC) -fpic -c cube.c

.c.o:
$(CC) $(CFLAGS) -c $<

clean:
/bin/rm -f *.o $(TARGET)
```