

const, structs, linked lists & compiling multi-file c programs



**KOÇ
UNIVERSITY**

COMP201 - Fall 2021 - Lab 5

const (recap)

const is used to declare **constants**

which means they are **NOT changeable after being created!**

- Function Declaration: ***int length(const char *str){...}***
// This function promises to not change str's characters
- Variable (Constant) Declaration: ***const int x = 10;***
// cannot modify this int
- Pointer Declaration: ***const int *arr = ...***
// cannot modify ints pointed to by arr
- Double Pointer Declaration: ***const char **strPtr = ...***
// cannot modify chars pointed to by *strPtr

Note: See the lecture on March 22 to learn more about these.

struct (recap) : a new variable type that is a group of other variables

```
// declaring a struct type
```

```
struct table {
```

```
// features of a table as a struct
```

```
// maybe useful for some furniture company
```

```
int width; //cm
```

```
int depth; //cm
```

```
};
```

```
...
```

```
struct table modelX; // construct structure instances
```

```
table.width = 120;
```

```
today.depth = 150;
```

```
struct table modelZ = {120, 150}; // shorter initializer syntax
```

struct (recap)

- **typedef**: a keyword used to avoid having to include the word struct every time you make a new variable of that type.
 - Usage:

```
typedef struct table{...} table_type;  
table_type t;
```
- If you pass a struct as a parameter, like for other parameters, C passes a copy of the entire struct.
- If you want to modify the struct in a function, pass a pointer of the struct to the function.
- The arrow operator lets you access the field of a struct pointed to by a pointer.
 - Usage: `t->width++;` // equivalent to `(*t).width++;`
- If a function returns a struct, it returns whatever is contained within the struct.
- **sizeof** gives you the entire size of a struct.
- You can use arrays of structs just like any other variable type.


Note: See the lecture on March 22 to learn more about these.

Linked List

A linear collection of data elements where **each element points (linked) to the next**. Each element is called a **node**. Each node contains some content and a reference (pointer) to the next node.

For example: Let's visualize each node like this (

content next

) and assume each node stores an integer so that we store a list of integers in a linked list. The list [5, 3, 1] will store it as follows: 

```
typedef struct node {  
    int content;  
    node* next;  
} node;
```

```
node node1, node2, node3;  
node1.content = 5;  
node1.next = &node2; // node1.next points to node2  
node2.content = 3;  
node2.next = &node3; // node2.next points to node3  
node3.content = 1;
```

```
printf("The value stored in the third node is %d", node.next->next->content); // prints 1
```

Linked List vs Array

- Insert/delete elements are generally cheaper for Linked List.
- Linked List use more memory than arrays because of the storage used by their pointers.
- Nodes in a linked list must be read in order from the beginning as linked lists are inherently sequential access.
- Difficulties arise in linked lists when it comes to reverse traversing.
- ...

Multi-file programs

A large C program should be divided into multiple files
because it is easier to manage and maintain multiple files than one huge file.

It also allows some of the code, e.g. utility functions, to be shared with other programs.

*e.g. Linux operating system has 50,000+ .c files
(<https://github.com/torvalds/linux>)*

Dividing by topic

Each file should contain a group of functions that do related things!

Examples:

- functions that do file or graphical input/output
- the set of functions that handle access to your database
- the implementation of an abstract data type, e.g. a linked list or a binary search tree
- a group of functions to do related numerical computations, e.g. a matrix manipulation package or a set of statistics functions

A large program might be divided into several such files, plus a main program file.

Compilation & Linking

Compilation means the processing of a source code file (.c) in order to create of an 'object' file that includes the machine language instructions that correspond to the source code file that was compiled.

Linking refers to the creation of a single executable file from multiple object files.

During compilation, if the compiler could not find the definition for a particular function, it would just assume that the function was defined in another file.

The linker, on the other hand, may look at multiple files and try to find references for the functions that weren't mentioned.

Compiling files

Let's say we have 3 files (file1.c, file2.c, file3.c), we can compile them like this:

gcc -c file1.c file2.c file3.c

GCC will generate corresponding object files (file1.o, file2.o, file3.o).

Linking files

Now, we have 3 object files (file1.o, file2.o, file3.o), we can link them like this:

gcc -o myprogram file1.o file2.o file3.o

GCC will generate an executable file called “myprogram”.

Compiling and linking files in one step

We can compile and link the files in one step like this:

gcc -o myprogram file1.c file2.c file3.c

GCC will generate an executable file called “myprogram”.

The difference between previous slide:

instead of the object files (.o), we give the source files (.c) to GCC

Function Declarations

Declarations (also called prototypes) are used when the compiler must be informed about a function at a point when it is inconvenient to give the compiler the full code for the function.

For example:

- the function is defined in one file but called in another file
- two functions call one another, recursively
- you want to reorder the functions within a file, e.g. to put the main function first

Declarations contain only type information for the function, but not its actual code.

See the following examples:

```
int min (int, int);           // no parameter names
double cbrt(double x);       // with a parameter name
```

Place of Function Declarations

A declaration for a function can be put in the file of code which calls the function,
typically at/near the top of the file.

Header Files

The main purpose of header files is to make definitions and declarations accessible to functions in more than one file.

For example, if functions from two files need to access the same global constant (e.g. a definition of the constant π), that variable should be defined in a header file.

Header File Example

There is a step called **preprocessing** before the compilation. In preprocessing, compiler processes statements like `#include`.

`#include "answer.h"` commands preprocessor to include the content of the "answer.h" file at the place where `#include "answer.h"` is written. (this is done in the preprocessing step)

Example: Include File

answer.h

```
int answer(double x);
```

answer.c

```
#include "answer.h"
int answer(double x) {
    return x * 21;
}
```

main.c

```
#include "answer.h"

int main(void) {
    printf("answer(2) = %d\n", answer(1));
    return 0;
}
```

Makefiles

Makefile is a program building tool which aids in simplifying building program executables that may need multiple source files.

Makefiles are special format files that help build and manage the projects automatically.

Makefile Rules

Makefile rules are like functions. The general syntax of a Makefile rule:

```
target [target...] : [dependent ....]  
    [ command ...]
```

<TAB>

An example of a Makefile rule:

```
myprogram: file1.c file2.c file3.c  
    gcc -o myprogram file1.c file2.c file3.c
```

Here, “myprogram” is the target. It depends on the existence of file1.c, file2.c and file3.c.

The command for this target is “gcc -o myprogram file1.c file2.c file3.c”.

When we command “**make myprogram**” in the directory where this Makefile is placed, it will first check the existence of file1.c, file2.c & file3.c and then it will run the given gcc command.

Makefile Macros

Makefile macros are like variables. They are defined in a Makefile as = pairs.

We can use the value of a defined macro by: `$(MACRONAME)`

An example has been shown below:

```
CC = gcc
CFLAGS = -Wall
MYFACE = ":*)"

myprogram: file1.c file2.c file3.c
    $(CC) $(CFLAGS) -o myprogram file1.c file2.c file3.c

printmyface:
    echo $(MYFACE)
```

Here, CC and CFLAGS are conventional macros used to refer to the compiler program and the compiler flags (options), respectively.

Makefile Conventions

You should always browse the Makefile before giving any make commands. However, it is reasonable to expect that the targets `all` (or just `make`), `install`, and `clean` is found.

make all – It compiles everything so that you can do local testing before installing applications.

make install – It installs applications at right places.

make clean – It cleans applications, gets rid of the executables, any temporary files, object files, etc.

These targets (`all`, `install`, `clean`) should be defined in the Makefile conventionally.

Complete Makefile Example

In this example;

- “CC” set to be GCC compiler
- “all” is set to depend on clean, install and run in the given order so that make will run the target in the order
- “install” is set to be alias for myprogram
- “clean” removes all object files and executables
- “run” runs the program
- “file1.o” and “file2.o” compiles file1.c and file2.c
- “myprogram” target links the object files (file1.o and file2.o)

```
CC = gcc

all: clean install run

install: myprogram

clean:
    rm -rf file1.o file2.o myprogram

run: myprogram
    ./myprogram

myprogram: file1.o file2.o
    $(CC) -o myprogram file1.o file2.o

file1.o: file1.c
    $(CC) -c file1.c

file2.o: file2.c
    $(CC) -c file2.c
```

More about Makefiles

There are many more helpful features such as:

- **suffix rules:** allows you to define targets for files with same suffix
- **conditional directives:** allows you to run commands based on conditions (like if statement)
- **include directive:** allows you to suspend reading the current makefile and read one or more other makefiles before continuing
- **override directive:** allows you to override the value of a macro

For details, see:

https://www.tutorialspoint.com/makefile/makefile_quick_guide.htm

<https://www.gnu.org/software/make/manual/make.pdf>

Acknowledgements and References

The slides are compiled and/or adapted from the following sources:

- <https://www.cs.hmc.edu/~geoff/classes/hmc.cs070.200401/notes/multi-file.html>
- https://cgi.cse.unsw.edu.au/~cs1511/19T1/lec/multiple_file_C/slides
- <https://www.cprogramming.com/compilingandlinking.html>
- https://www.tutorialspoint.com/makefile/makefile_quick_guide.htm
- <https://www.gnu.org/software/make/manual/make.pdf>
- https://en.wikipedia.org/wiki/Linked_list