Const, Structs, Linked Lists & Makefiles

KOÇ UNIVERSITY

COMP201 - Lab 5

Spring 2024

const

- Used to declare ***constants*****,** which means they are ***immutable*** (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 **x**.
  - Pointer Declaration: **const int *arr = …**

    // Cannot modify ints **arr** points to.
  - Double Pointer Declaration: **const char **strPtr = …**

    // Cannot modify chars ***strPtr** points to.

struct

- A new variable type that is a group of other variables.

```
// Declare a user-defined type.
struct table {
// Multiple variables together into a single aggregate data type.
// Maybe useful for a furniture company.
int width; // cm
int depth; // cm
};
...
struct table modelX;
modelX.width = 120;
modelX.depth = 150;
...
// OR
struct table modelZ = {120, 150};
```

struct

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

```
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.

```
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.

Linked List

- A linear collection of data elements called **nodes**. Each node stores some content and a reference (pointer) to the next node.
- Usage of doubly linked list (each node stores the address of previous node as well) and circular linked list (last node points to the first node) depending on the specific requirements of the application.

Assume each node stores an integer so that we store a list of integers [5,3,1] in a singly linked list.

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

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

Linked List (Singly) vs Array

- Insertion & deletion operations are generally faster for ***Linked List.***

- ***Linked List*** uses more memory than arrays because of the storage used by their pointers.

- ***Nodes*** in a linked list must be read sequentially.

- Difficulties arise in singly linked lists when it comes to reverse traversing.

Working with Multiple Files

- 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)

Working with Multiple Files - 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.

Working with Multiple Files - Compilation & Linking

- ***Compilation*** means the processing of a source code file (.c) in order to create 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.

Working with Multiple Files - Compilation

- 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).

Working with Multiple Files - Linking

- 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***.

Working with Multiple Files - Combining Compilation & Linking

- 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***.

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, or the ***definition***, of the function.

  For example:

  - The function is defined in one file but called in another file.
  - Two functions call one another.
  - 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.

```
int min (int, int);

double cbrt(double x); // with parameter names (preferred).
```

Function Declarations - Header Files

- 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.

- 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 **π**), that variable should be defined in a header file.

Function Declarations - Header File Example

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

- `#include “answer.h”` in *Figure 1* 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;
}
```

Figure 1

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.

Makefiles - Rules

- General syntax:

```
target [target...] : [dependent ....]
<TAB> [ command ...]
```

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

Figure 2

- **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.

Makefiles - Macros

- **Macros** are like variables. They are defined in a Makefile as = pairs. We can use the value of a defined macro by: `$(MACRONAME)`.

```
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)
```

Figure 3

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

Makefiles - 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 the 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.

Makefiles - Complete Example

```
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
```

Figure 4

- `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”.

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.html

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.html
- https://www.gnu.org/software/make/manual/make.pdf
- https://en.wikipedia.org/wiki/Linked_list