

COMP 201

Spring 2026



KOÇ
UNIVERSITY

Lab 4

Constants, Structs, Linked Lists, and Makefiles

const

- Define **constants** in your program (not changeable after created)

- Variable declaration:

```
const int x = 0 // Cannot modify x
```

- Pointer declaration:

```
const int *arr = ... // Cannot modify ints that arr points to
```

- Double pointer declaration:

```
const char **strPtr = ... // Cannot modify chars that *strPtr points to
```

- Function declaration:

```
int length(const char *str) {...} // Won't change strings characters
```

struct

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

```
struct point {  
    int x;  
    int y;  
};
```

```
struct point point_1;  
point1.x = 0;  
point1.y = 0;
```

```
// You can combine declaration and assignments  
struct point point2 = {1, 2};
```

struct - typedef

- **typedef**: creates a new type name that refers to the same struct type, so you can avoid having to write struct everytime.

```
typedef struct point {  
    int x;  
    int y;  
} point;
```

// Instead of “struct point point_1” you can write:

```
point point_1;  
point1.x = 1  
point1.y = 2
```

```
point point2 = {2, 3};
```

Note: typedef is not limited to structs. You can define new type names for other types too:

```
typedef int* int_ptr;
```

struct - Copy by assignment

- You can assign the value of a struct from a struct of the same type.
- It will copy the entire content.

```
typedef struct point {  
    int x;  
    int y;  
} point;
```

```
point p1 = {0, 1};  
point p2 = p1;
```

```
p2.x = 5;  
printf("%f", p1.x); // prints 0
```

struct as a Parameter

- If you pass a struct as a parameter, it passes a copy of the struct.

```
void increase_x(point p) {  
    p.x++;  
}  
  
...  
point point2 = {2, 3};  
increase_x(point2)  
printf("%d", point2.x) // prints 2
```

struct as a Parameter

- If you want to modify the struct, pass a pointer of the struct to the function.

```
void increase_x(point *p) {  
    (*p).x++;  
}  
  
...  
point point2 = {2, 3};  
increase_x(&point2)  
printf("%d", point2.x) // prints 3
```

struct

- The **arrow (->) operator** lets you access the field of a struct via its pointer:

```
point1_ptr->x++; // equivalent to (*point1_ptr).x++;
```

- If a function returns a struct, it returns a copy of it (return by value):

```
point create_origin() {  
    point origin = {0,0};  
    return origin; // returns a copy of origin with x=0, y=0  
}
```

- **sizeof** gives you the entire size of struct:

```
sizeof(point) // 8
```

- You can use arrays of structs like any other variable type:

```
point points[3]; // array of 3 struct points
```

struct - Dynamic Allocation

- You can **malloc** and **free** structs, just like other data type.

```
typedef struct point {  
    int x;  
    int y;  
} point;
```

```
point* allocate_point(int x, int y) {  
    point* new_point = (point*) malloc(sizeof(point));  
    new_point->x = x;  
    new_point->y = y;  
    return new_point;  
}
```

struct - Pass Copy or Pointer

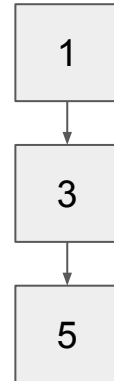
- **Value passed:** Passing a pointer and is cheaper and takes less space unless struct is small.
- **Field access:** Indirect access through pointers (->) are a bit more expensive and can be harder for compiler to optimize.
- For small structs, passing a copy of the struct can be faster and often preferred if function only reads data; for large structs use pointers.

Linked Lists

- A sequence of nodes arranged linearly. Each node contains some data and a reference to the next node in the sequence.
- **Doubly linked list:** each node also contains a reference to the previous node.
- **Circular linked list:** last node points to the first node
- Example linked list implementation:

```
typedef struct node {  
    int number;  
    node* next;  
} node;  
node node1, node2, node3;
```

```
node1.number = 5;  
node1.next = &node2;  
node2.number = 3;  
node2.next = &node3;  
node3.number = 1;
```



(Singly) Linked Lists vs Arrays

- Insertion and deletion operations are generally faster for linked lists.
- Linked lists use more memory per element because each node stores additional pointer fields alongside the data
- Nodes in a linked list must be read sequentially
- Singly linked lists make reverse traversal difficult because nodes do not store points to their predecessors

Working with Multiple Files

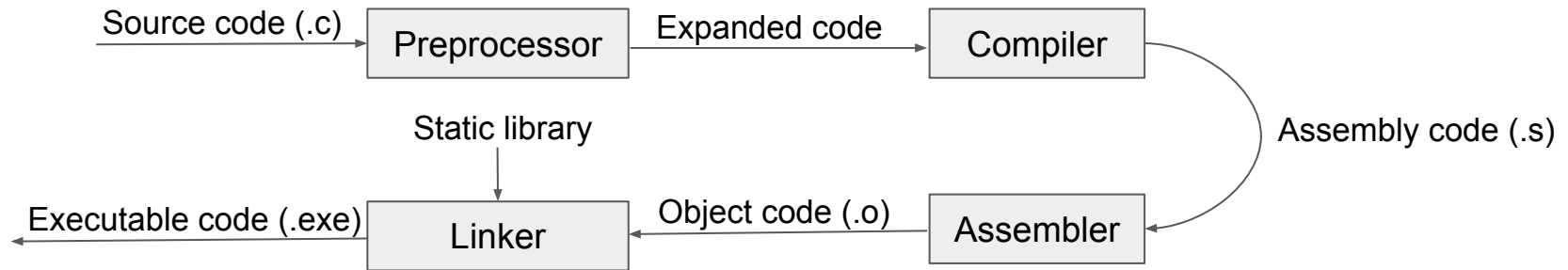
- A large C program should be split into multiple source files because this improves manageability and maintainability.
 - It also enables reusable components, such as utility functions, to be shared with other programs.
- For example, the Linux kernel has more than 50,000 C files:
<https://github.com/torvalds/linux>

Working with Multiple Files - Dividing by Topic

- Each file should group together functions that perform related tasks.
- Examples include:
 - Functions for file or graphical input/output
 - Functions that manage access to a database
 - Implementations of abstract data types, such as linked lists or binary search trees
 - Groups of related numerical routines, such as matrix-manipulation functions or statistical utilities
- A large program may consist of several such files, along with a separate main program file.

Working with Multiple Files - Compilation & Linking

- **Compilation** is the process of translating a source file (.c) into an object file containing the machine instructions generated from that source.
- **Linker** combines multiple object files to produce a single executable.
- During compilation, if the compiler encounters a function that is declared but not defined, it assumes the definition exists in another file.
- The **linker** then resolves these external references by searching the object files and libraries for the missing function definitions.



Working with Multiple Files - Compilation & Linking

- Let's say we have 3 files: `file1.c`, `file2.c`, `file3.c`. we can compile them with:

```
$ gcc -c file1.c file2.c file3.c
```

- GCC will generate corresponding object files: `file1.o`, `file2.o`, `file3.o`
- We can link these object files with:

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

- GCC will generate an executable file with name `myprogram`.
- We can also compile and link with a single command:

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

Function Declarations

- Declarations (function prototypes) are used when the compiler needs to know about a function before its full definition is available. This is necessary in situations such as:
 - A function defined in one file is called from another file
 - Two functions call each other (mutual recursion)
 - You want to reorder functions within a file, for example placing `main` function at the top
- Declarations provide only the function's type information, not its implementation.

For example:

```
int min(int, int);  
double cbrt(double x); // including parameter names is preferred
```

Function Declarations - Header Files

- A function's declaration can be placed in the source file that calls it, typically near the top.
- **Header files** exist to make declarations and definitions accessible to multiple source files.
- If multiple source files need to use the same global constant, then:
 - The **declaration** goes in the header file:

```
extern const double PI;
```
 - The **definition** goes in exactly one .c file:

```
const double PI = 3.141592653589793;
```

Function Declarations - Header Files

- During preprocessing, the compiler expands `#include` by inserting the contents of the header file.
- This lets each source file see the declaration it needs.

square.h

```
double square(double x);
```

square.c

```
#include "square.h"
```

```
double square(double x) {
```

```
    return x * x;
```

```
}
```

main.c

```
#include "square.h"
```

```
int main(void) {
```

```
    printf("%d", square(5))
```

```
    return 0;
```

```
}
```

Makefiles

- Makefile is a build tool that automates compiling programs composed of multiple source files.
- A Makefile is a specially formatted file that defines how a project should be built and rebuilt automatically.
- Programmers like to automate repetitive tasks.
- You don't want to retype the same command every time:

```
gcc -Wall -Werror -std=c17 main file1.c file2.c file3.c
```
- You don't want to recompile everything every time you change something (especially if you have many files of source code)

Makefiles

- Basic idea:
 - Scripts for executing commands
 - Dependencies to avoid unnecessary work

- General syntax:

```
target : dependencies
```

```
<TAB> command
```

- `target`: the file you want to build or a symbol that refers to an action
- `dependencies`: the files/targets that the target depends on
- `command`: the shell command used to build the target (It can be any shell command). It must start with a TAB.
 - You can split commands over multiple lines by ending lines with `\`

Makefiles - Running Your Makefiles

- You should name your makefile “Makefile” or “makefile”.
- You can run your makefile with “`$ make <targetname>`” command in the terminal in the same directory that your makefile is located.
- You can also use other names for your makefile, but you should use this command to run it:

```
$ make -f <filename> <targetname>
```

Makefiles - Example

- Example:
 - myprogram: file1.c file2.c file3.c
`gcc -o myprogram file1.c file2.c file3.c`
 - myprogram is the target
 - It depends on file1.c, file2.c and file3.c. If any of these files change, myprogram must be rebuilt.
 - Command `gcc -o myprogram file1.c file2.c file3.c` tells make how to build the target from the dependencies
 - When you run `$ make myprogram` command, make checks:
 1. Does the file myprogram exist?
 2. Is its timestamp older than any of the dependencies?
 3. If yes, run the provided command to rebuild the target

Makefiles - Phony Targets

- A make target whose command does not create a file of the target's name
 - As long as target file doesn't exist, the commands will be executed because the target must be "remade".
- For example, target `clean` is a convention to remove generated files to "start over" from just the source:

```
clean:
```

```
    rm file1.o file2.o main
```

- Since `clean` is just an action, there is usually no file named `clean`, you can mark it as phony:

```
.PHONY: clean
```

- Otherwise, if a file named `clean` exists, `$ make clean` may do nothing.

Makefiles - Variables

- Makefile variables (sometimes called macros) behave like C #define macros.
- Variables are defined with name = value pairs.
- To use a macro, reference it with: \$(MACRONAME)
- Example:

```
CC      = gcc
```

```
CFLAGS = -Wall
```

```
MYFACE = ":*"
```

```
myprogram: file1.c file2.c
```

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

```
printmyface:
```

```
    echo $(MYFACE)
```

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

Makefiles - Conventions

- Before running any make command, it is useful to inspect the Makefile. It is common to find the following conventional targets:
 - **make all** : Builds the entire project. This is typically used for compiling everything needed for local testing.
 - **make install** : Copies the built files to their installation locations.
 - **make clean** : Removes build artifacts such as executables, object files, and temporary files.
- These standard targets (all, install, clean) are conventionally included in Makefiles so users know what to expect.

Makefiles - Complete Example

```
CC = gcc
CFLAGS = -Wall -std=c99

.PHONY = all clean

all: program

clean:
    rm -f *.o program

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

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

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

- **CC** is set to the compiler program (gcc).
- **CFLAGS** is set to “-Wall -std=c99” and is used for compilation flags.
- **all** depends on the target **program**. It is marked as a phony target.
- **clean** removes generated files (file1.o, file2.o, and myprogram). It is marked as a phony target.
- The rule for **myprogram** builds the executable from the object files file1.o and file2.o.
- The rules for **file1.o** and **file2.o** compile file1.c and file2.c into object files.

Makefiles - Special Variables

- `$$` : for target name
- `$$` : for all sources
- `$$` : for leftmost source

- Examples:

```
program: file1.o file2.o
```

```
    gcc -o $$ $$
```

```
compile: file1.c header1.h header2.h
```

```
    gcc -c $$
```

More about Makefiles

- Makefiles support many additional useful features, including:
 - **Suffix rules:** Define rules that apply to all files with a given suffix (e.g., `.c` → `.o`).
 - **Conditional directives:** Allow Makefiles to execute different rules based on conditions (similar to `if` statements).
 - **Include directive:** Allows one Makefile to include others, letting you split build logic across multiple files.
 - **Override directive:** Allows overriding the value of a macro, even if it was set on the command line.
- **Further reading:**
 - 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
- https://courses.cs.washington.edu/courses/cse333/26wi/lectures/08/08_make_26wi.pdf
- https://courses.cs.washington.edu/courses/cse333/26wi/lectures/04/04_heap-structs_26wi.pdf