Announcement

- Homework 2 is posted!
- A written assignment, so no late days!
- Directly work on GradeScope.

Heads-up

- Tomorrow is Lunar new year eve to the new year (HK Time)
- Office hours canceled.
- Happy Lunar new year!

Today's plan

- Class interactions
  - Ex 3-1
  - Keys points
  - Recap discussion
- Class exercises
  - Ex 3-2

Ex 3-1

Volunteers?

- You practiced how to declare and define functions in C.
- You wrote functions using recursion.
  - Base cases (stopping criteria).
  - Divide into sub-problems.
  - Recursive call to deal with the sub-problems.
- You tried passing arrays to functions.
- You found you can change the arrays in the functions.
- You should notice that you CANNOT change the value of a single variable that you passed in the functions. i.e. the value of that variable in the caller is unchanged, even though you have changed it in the callee.
- How do I change the value of a variable in a function? Use pointers. We will come to that next week.

Key points - header files

- Header files (`.h`): we put declarations in header files.
- Source files (`.c`): we put definitions in source files. Why?
- Reduce dependency - faster compilation
  - We group similar concepts using header files.
  - It's likely when you make changes to a concept, you only modify a small set of header and source files.
  - Therefore, you only need to recompile a small set of files.
  - i.e. if you find you need to recompile lots of file for a small change, rearrange your grouping.
- Encapsulate your source codes - hide them from others
  - It is intended to separate the definitions from the declarations.
  - If you want to let others to use your libraries without knowing your own implementations, you give them the compiled libraries (`.lib`) and the header files.
  - Then, they can develop codes using your declarations and your 'compiled' implementations.
- These concepts become 'classes' later in C++.

Key points - separate compilation

- Similar concepts are grouped into sets of header/source files.
- Let us say we have group A, B, C.
- We compile each group to an object file, then we link them.
- Why do we need to link them?
  - In group A, we may call `Foo` implemented in group B.
  - We include the header file of `Foo`, so we can it in A.
  - When compiling group A, the compiler sees `Foo`'s declaration (in the header file), but not its definition.
  - The compiler puts a 'placeholder' for Foo and waits until later to fill in what it should be.
  - When compiling group B, the compiler sees `Foo`'s definition and compiled it in B's object file.
  - So, we need linking to link Foo's definition in B and Foo's placeholder in A.

Key points - header guards

- Recall: C allows duplicated declarations, but not definitions.
- Ideally, header files only contains declarations. Yet, in reality, we put definition in it too. e.g.
  - inline functions
  - `struct`, which we will dive into it soon
  - Additional note: in C++, we have forward declaration to further reduce dependency between header files.
- Then, problems come when we include this kind of header files twice.
- We define the same functions twice, which causes a "redefinition" compilation error
- That's why we introduce the header guard - a preprocessor step to avoid duplicated inclusion of a header file.

Key points - symbol redefinition in separate compilation

```c
// a.h
#ifndef A_H
#define A_H

int A();

#endif
```

```c
// a.c
#include "a.h"
#include "c.h"

int A() {
    C();
    return 0;
}
```

```c
// b.h
#ifndef B_H
#define B_H

int B();

#endif
```

```c
// b.c
#include "b.h"
#include "c.h"

int B() {
    C();
    return 1;
}
```

```c
// c.h
#ifndef C_H
#define C_H
#include <stdio.h>

void C() {
    printf("Hello\n");
}

#endif
```

```c
// main.c
#include "a.h"
#include "b.h"

int main() {
    A();
    B();
    return 0;
}
```

Key points - symbol redefinition in separate compilation

- Let us have two groups, A and B. Both include the same header file which defines an inline function `Bar`
  - When compiling group A, the compiler sees the `Bar`'s definition, so it compiles `Bar` in A's object file.
  - When compiling group B, it also sees it, so it also compiles `Bar` in B's object file.
  - Ops... we have two definitions of `Bar` now.
  - We have no problems when compiling them. But we will receive a 'symbol redefinition' linking error.
  - This type of linking error cannot be resolved by header guards.
- In this case (for C99), use `static inline` to force the compiler treat it as a static inline function.

Key points - symbol redefinition in separate compilation

```c
// a.h
#ifndef A_H
#define A_H

int A();

#endif
```

```c
// a.c
#include "a.h"
#include "c.h"

int A() {
    C();
    return 0;
}
```

```c
// b.h
#ifndef B_H
#define B_H

int B();

#endif
```

```c
// b.c
#include "b.h"
#include "c.h"

int B() {
    C();
    return 1;
}
```

```c
// c.h
#ifndef C_H
#define C_H
#include <stdio.h>

static inline void C() {
    printf("Hello\n");
}

#endif
```

```c
// main.c
#include "a.h"
#include "b.h"

int main() {
    A();
    B();
    return 0;
}
```

Key points - Makefiles

- Now, using separate compilation, we need to type the 'gcc' commands multiple times with different source files
- We also want to compile only those we have make changes.
- Makefiles help us to do that.
  - It specifics the rules of complication and linking
  - A 'target' is an object/executable/command that we want to compile/link/run
  - After the 'target', it's a list of files that the target depends on.
  - Next line followed by a 'tab' is the command to run to 'make' the target.
  - We can define variables in Makefiles and use them with the dollar bracket, i.e. `$(var_name)`.
- Userful automatic variables:
  - `$@`: The name of the target
  - `$<`: The first dependency of the target
  - `$^`: All dependency of the target

Key points - Makefiles

```
1   CC=gcc
2   CFLAGS=-std=c99 -pedantic -Wall -Wextra
3
4   ifdef ENABLE_GDB
5   CFLAGS+=-g
6   endif
7
8   main: main.o a.o b.o
9           $(CC) -o $@ $^
10
11  main.o: main.c c.h
12          $(CC) $(CFLAGS) -c $<
13
14  a.o: a.c a.h c.h
15          $(CC) $(CFLAGS) -c $<
16
17  b.o: b.c b.h c.h
18          $(CC) $(CFLAGS) -c  $<
19
20  clean:
21          rm -f *.o main
22
23  rebuild: clean main
```

Why do we need header guards?

Mentimeter

- Avoid duplicated inclusion of a header file
- To prevent multiple definitions of a function.
- so that we can prevent duplicate definitions
- to prevent duplicate definitions
- They prevent duplicate definitions
- To avoid multiple definition

The correct answer is: To avoid redefinition compilation errors caused by including the header files multiple times.

What is the difference between compiling and linking?

Mentimeter

- compiling give .o file, linking gives executable ✗
- Compiling converts source code to object code and linking takes all the object code to make it into a single executable ✗
- Compiling compiles each individual source code file. Lininking links these object files into a single .out ✗
- compiling just runs a single file and linking runs multiple files and makes one executable ✗
- compiling makes an executable and linking connects all of the executables ✗
- Compiling turns source code into object code, linking puts source code into one file. ✗

The correct answer is: Compiling compiles source files to object files while linking links object files to executable.

6

What compiler flag do we use to create object files and what extension do those files have?

Mentimeter

-c .o ✕ 2x

- o, .o files ✕

-c and .o ✕

-c -o ✕

-o .o ✕

The correct answer is: `-c', `.o'

6

What is a target in a Makefile?

Mentimeter

- object/executable/command that we want to compile/link/run
- A file
- the final file to run
- a function
- The file we want to generate from source files

The correct answer is: Something that we want to `make'. It could be an object file, an executable, or even just a command to run.

5

What are the advantages of using Makefiles?

Mentimeter

Only compile files we have changed, Separate compilation ✕

You do not have to recompile every source file, only the ones changed. Also faster, less typing ✕

they allow you to link separate files in dependencies so you don't individually need to compile each one ✕

keeps track of compiling dependencies for you and only compiles what's been changed ✕

Simplify our commands ✕

The correct answer is: It helps us to keep track of which files need to be recompiled. It's a long-term laziness. Once rules are specified, it works for the long term.

5

Q & A

Mentimeter

0 questions

0 upvotes

Class exercises

Ex 3-2