Organizing Large C Programs

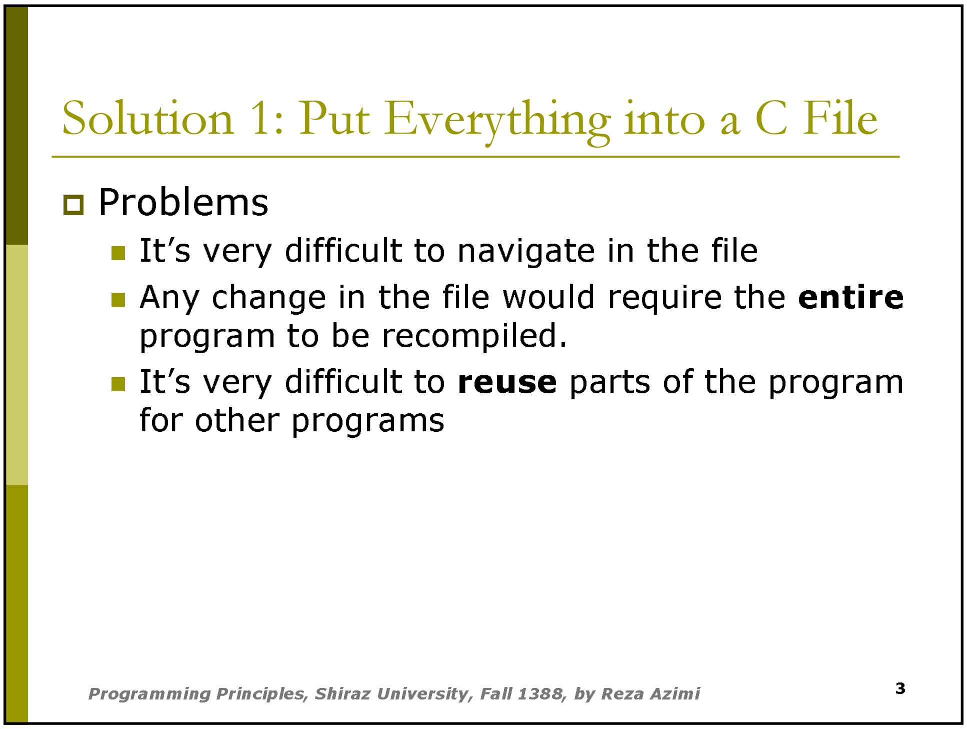
Reza Azimi

Principles of Programming in C

Department of Electrical and Computer Engineering
Shiraz University
Fall 1388

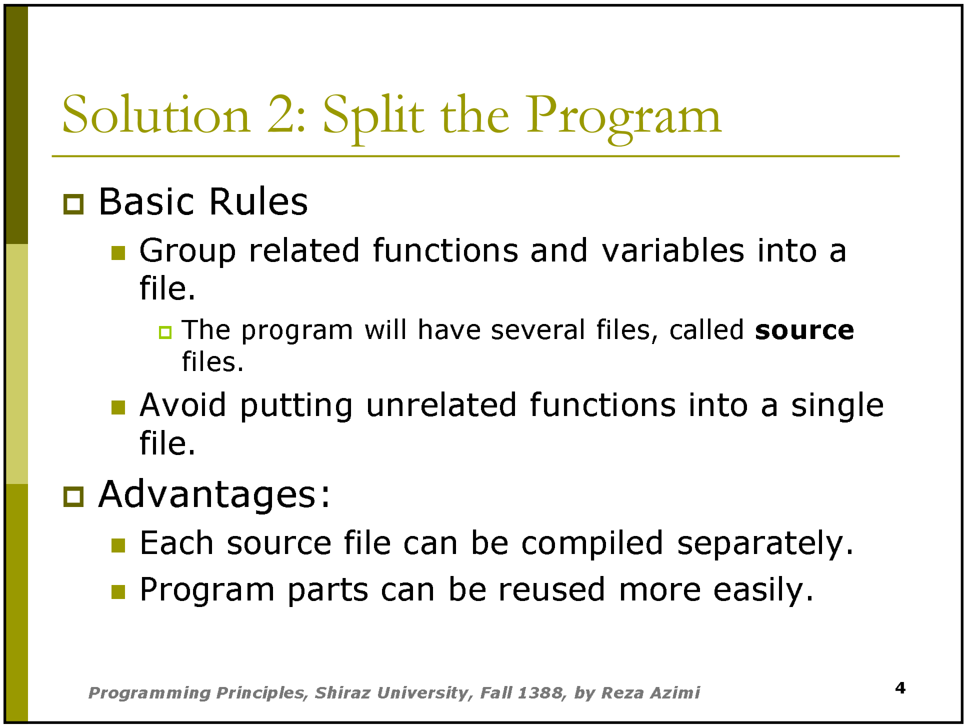
Problem

- Programs can become quite large
  - Linux Operating System is around 8 million lines of C code.
  - Database Management Systems (examples: Oracle and DB2) are tens of million lines of code.
  - Compilers (including gcc) are usually several million lines of code.
    - BTW, gcc is written in C!
  - Java Virtual Machines (mostly written in C and C++) are at least several hundred thousand of lines of C code.
- **How should we organize large C (or any other language) programs.**



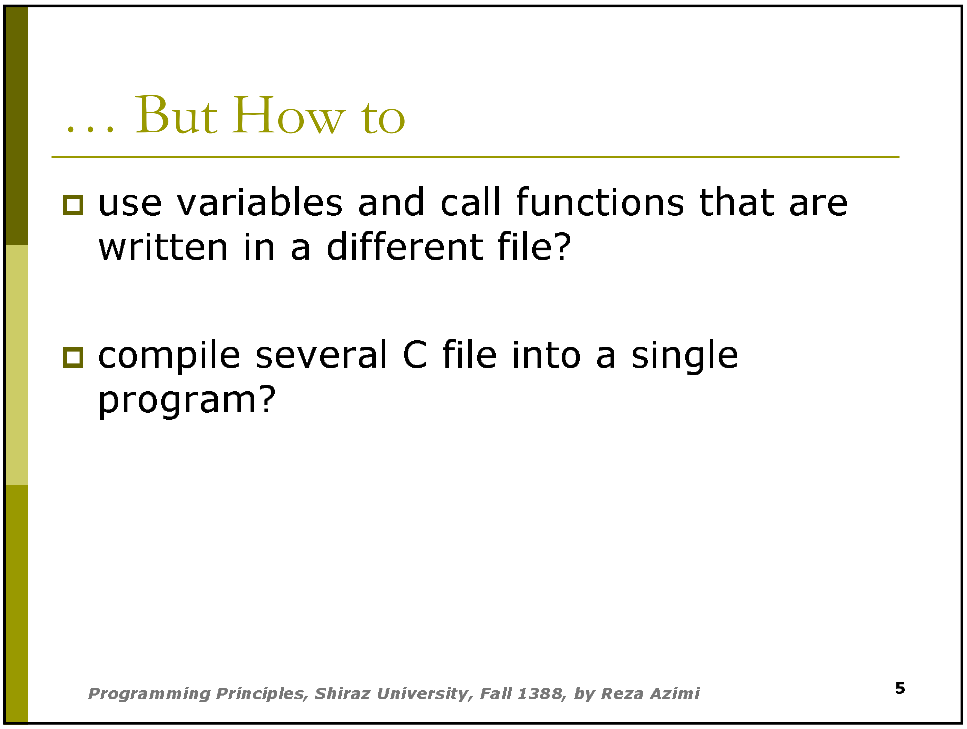
Solution 1: Put Everything into a C File

- Problems
  - It's very difficult to navigate in the file
  - Any change in the file would require the **entire** program to be recompiled.
  - It's very difficult to **reuse** parts of the program for other programs



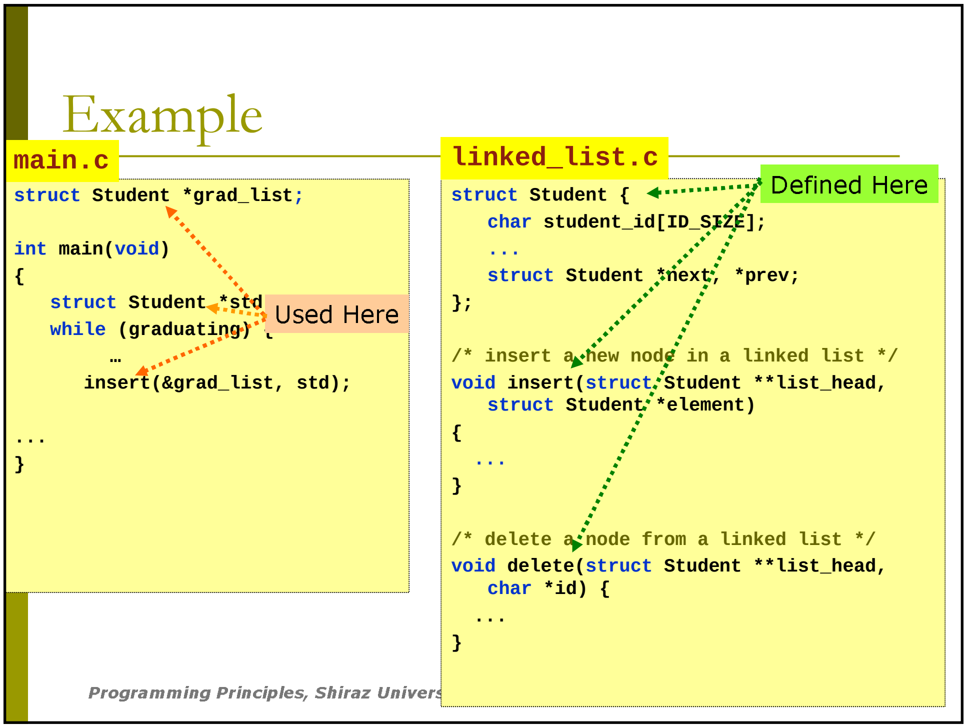
Solution 2: Split the Program

- Basic Rules
  - Group related functions and variables into a file.
    - The program will have several files, called **source** files.
  - Avoid putting unrelated functions into a single file.
- Advantages:
  - Each source file can be compiled separately.
  - Program parts can be reused more easily.



… But How to

- use variables and call functions that are written in a different file?
- compile several C file into a single program?



Example

main.c

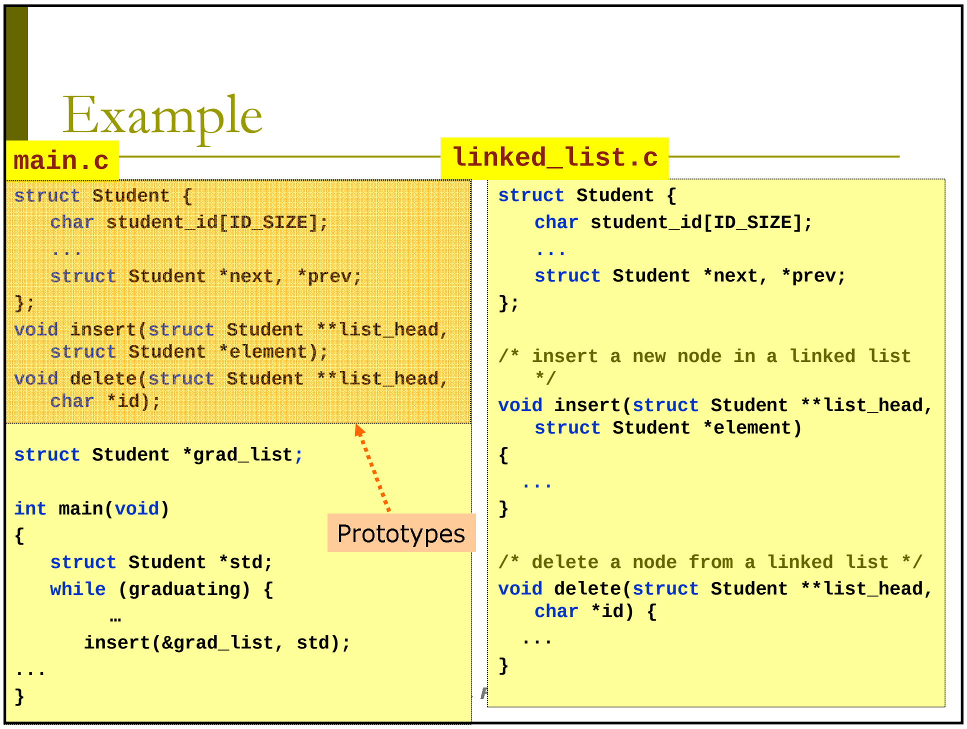
```
struct Student *grad_list;

int main(void)
{
   struct Student *std
   while (graduating) {
        …
      insert(&grad_list, std);

...
}
```

Used Here

linked_list.c

```
struct Student {
   char student_id[ID_SIZE];
   ...
   struct Student *next, *prev;
};

/* insert a new node in a linked list */
void insert(struct Student **list_head,
   struct Student *element)
{
  ...
}

/* delete a node from a linked list */
void delete(struct Student **list_head,
   char *id) {
  ...
}
```

Defined Here



Example

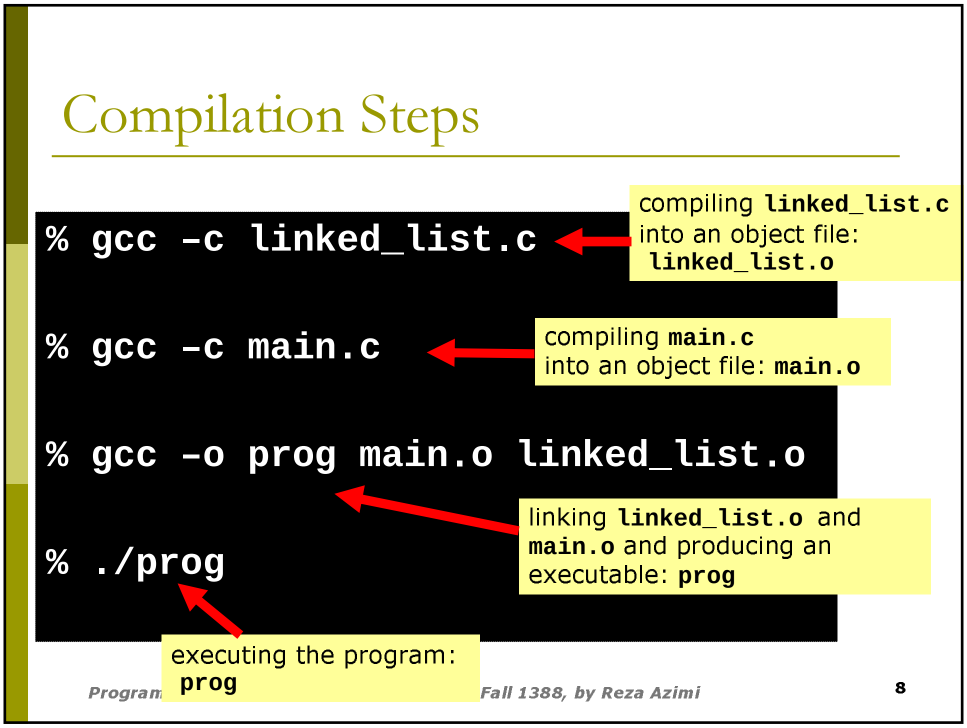
main.c

```
struct Student {
    char student_id[ID_SIZE];
    ...
    struct Student *next, *prev;
};
void insert(struct Student **list_head,
    struct Student *element);
void delete(struct Student **list_head,
    char *id);

struct Student *grad_list;

int main(void)
{
    struct Student *std;
    while (graduating) {
        …
       insert(&grad_list, std);
...
}
```

Prototypes

linked_list.c

```
struct Student {
    char student_id[ID_SIZE];
    ...
    struct Student *next, *prev;
};

/* insert a new node in a linked list
    */
void insert(struct Student **list_head,
    struct Student *element)
{
  ...
}

/* delete a node from a linked list */
void delete(struct Student **list_head,
    char *id) {
  ...
}
```



Compilation Steps

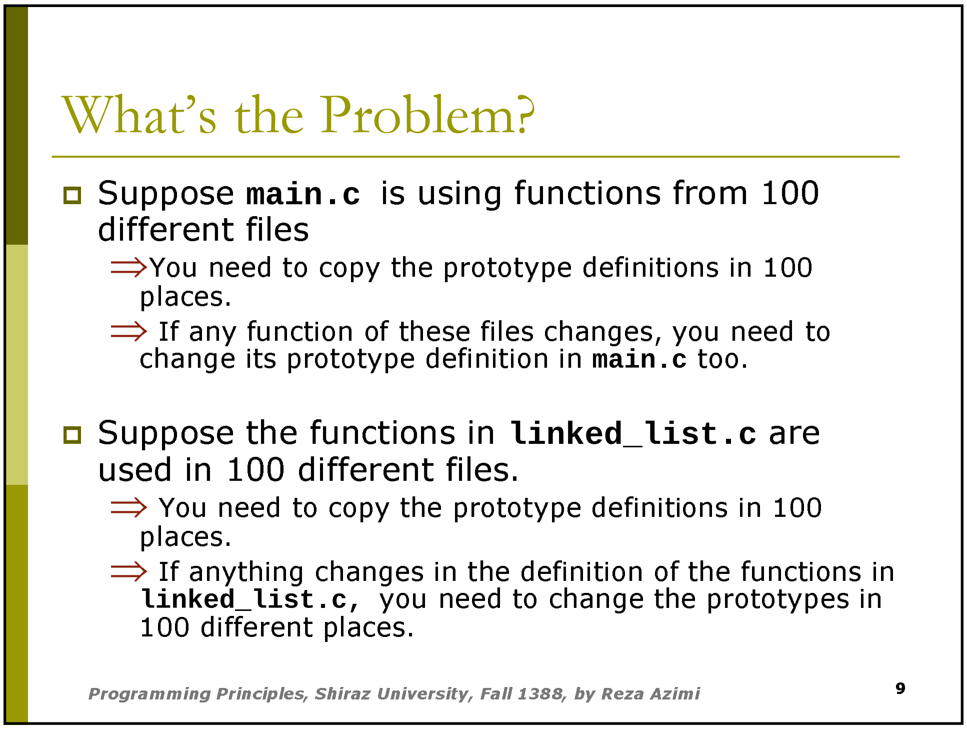
```
% gcc -c linked_list.c

% gcc -c main.c

% gcc -o prog main.o linked_list.o

% ./prog
```

compiling **linked_list.c** into an object file: **linked_list.o**

compiling **main.c** into an object file: **main.o**

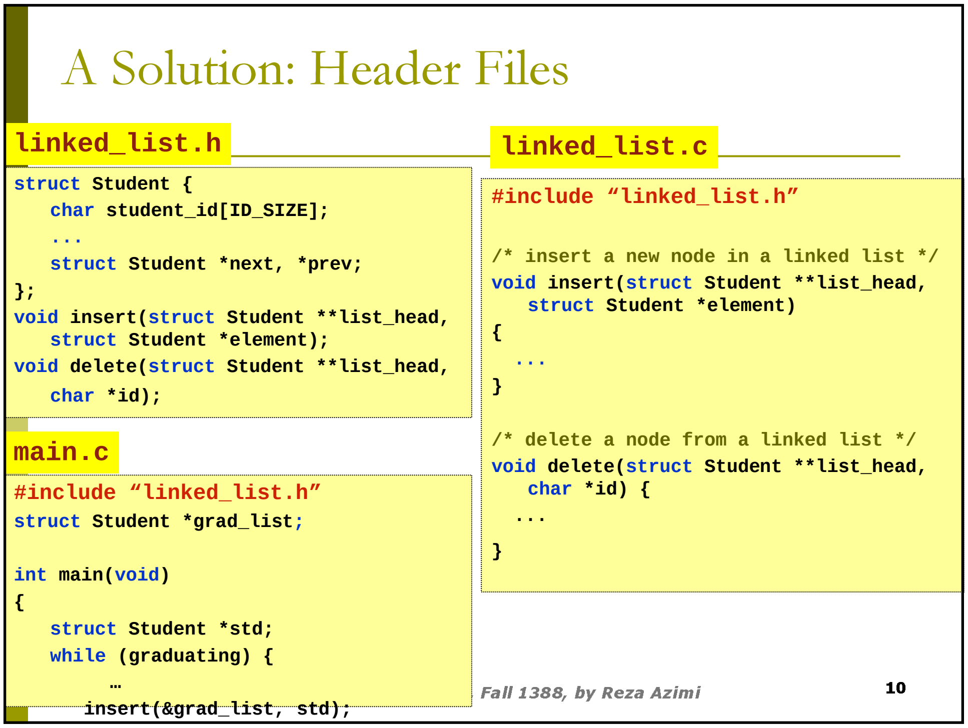
linking **linked_list.o** and **main.o** and producing an executable: **prog**

executing the program: **prog**



What’s the Problem?

- Suppose **main.c** is using functions from 100 different files
  - ⇒You need to copy the prototype definitions in 100 places.
  - ⇒ If any function of these files changes, you need to change its prototype definition in **main.c** too.
- Suppose the functions in **linked_list.c** are used in 100 different files.
  - ⇒ You need to copy the prototype definitions in 100 places.
  - ⇒ If anything changes in the definition of the functions in **linked_list.c,** you need to change the prototypes in 100 different places.



A Solution: Header Files

linked_list.h

```
struct Student {
   char student_id[ID_SIZE];
   ...
   struct Student *next, *prev;
};
void insert(struct Student **list_head,
   struct Student *element);
void delete(struct Student **list_head,
   char *id);
```

main.c

```
#include “linked_list.h”
struct Student *grad_list;

int main(void)
{
   struct Student *std;
   while (graduating) {
        …
      insert(&grad_list, std);
```

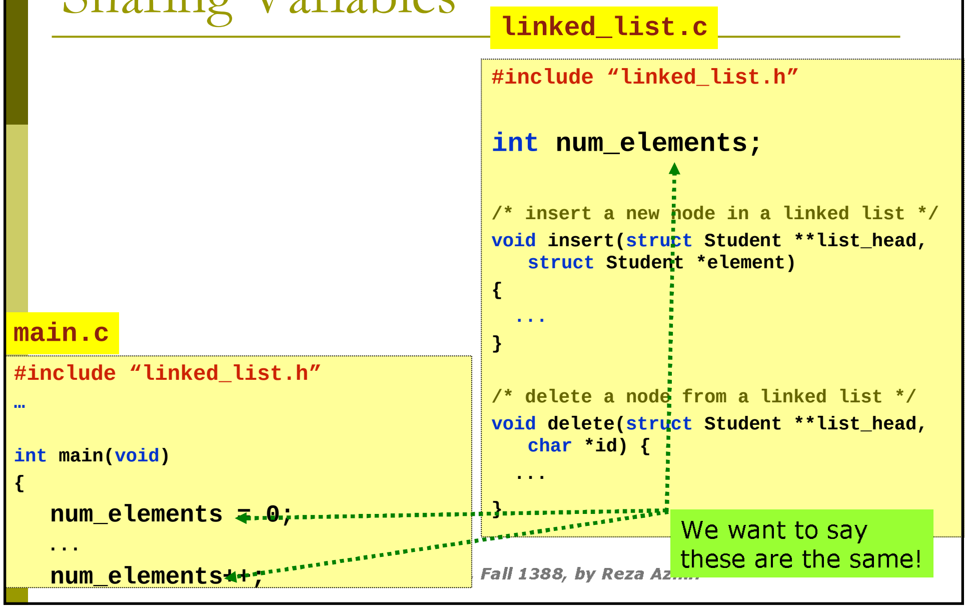
linked_list.c

```
#include “linked_list.h”

/* insert a new node in a linked list */
void insert(struct Student **list_head,
   struct Student *element)
{
  ...
}

/* delete a node from a linked list */
void delete(struct Student **list_head,
   char *id) {
  ...
}
```

#include “file.h”

- A ***preprocessor*** directive
  - is activated **before** compilation
- Pastes the entire content of **“file.h”,** wherever the **#include** appears.
- **#include <file.h>** looks for system header files first, so use “” for your own header files.
- Now, we know what means to say
  #include <stdio.h>

Sharing Variables



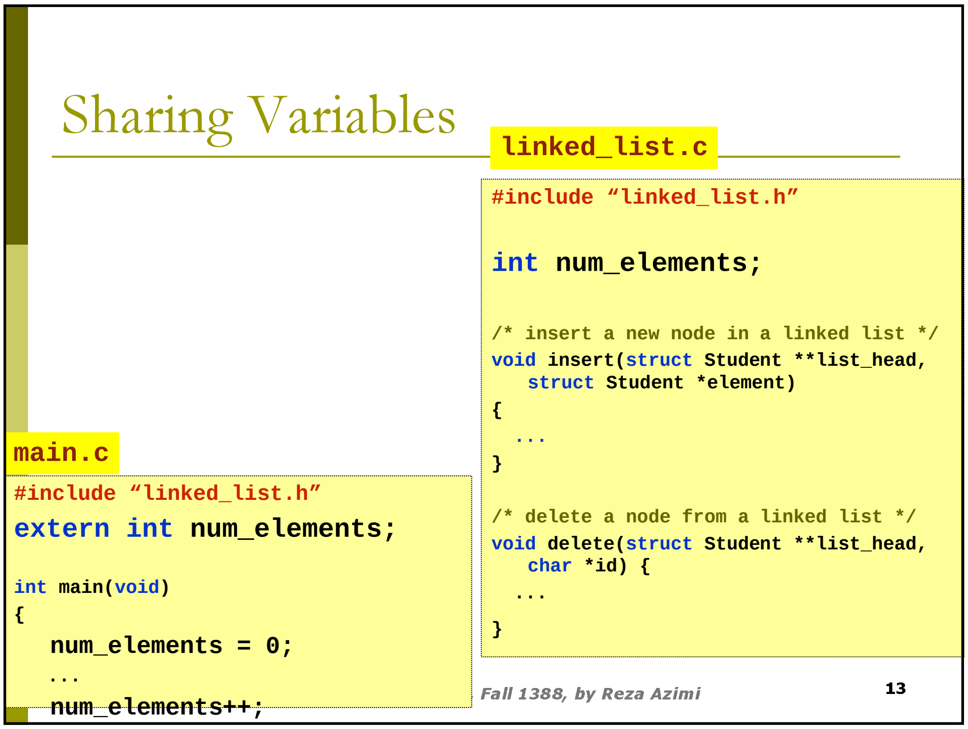
linked_list.c

```
#include “linked_list.h”

int num_elements;

/* insert a new node in a linked list */
void insert(struct Student **list_head,
    struct Student *element)
{
  ...
}

/* delete a node from a linked list */
void delete(struct Student **list_head,
    char *id) {
  ...
}
```

main.c

```
#include “linked_list.h”
…

int main(void)
{
   num_elements = 0;
   ...
   num_elements++;
```

We want to say these are the same!



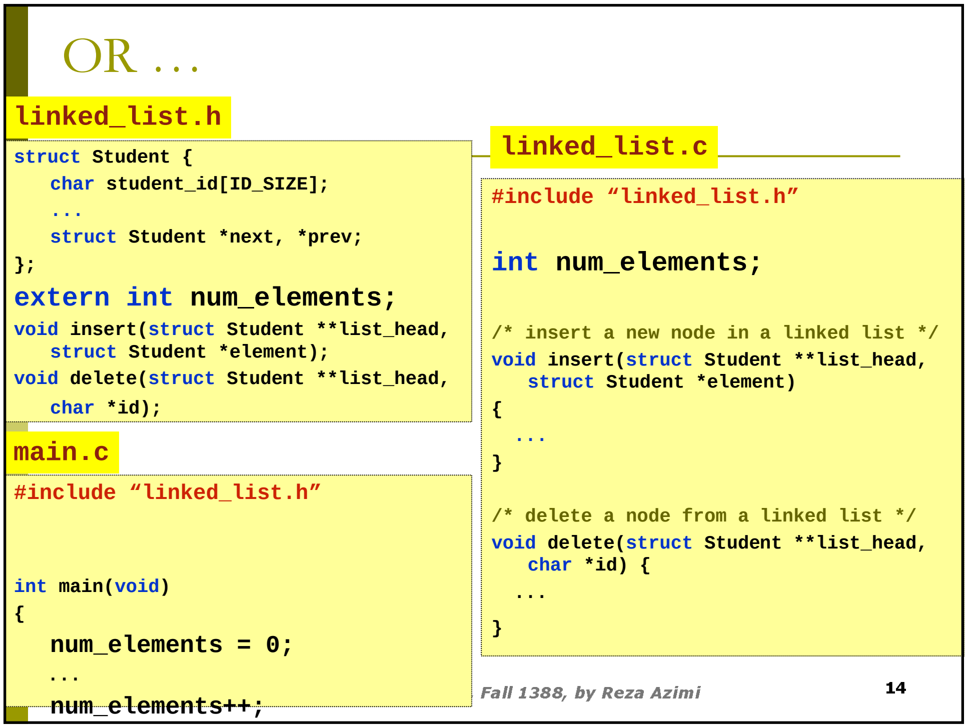
Sharing Variables

linked_list.c

```
#include “linked_list.h”

int num_elements;

/* insert a new node in a linked list */
void insert(struct Student **list_head,
    struct Student *element)
{
  ...
}

/* delete a node from a linked list */
void delete(struct Student **list_head,
    char *id) {
  ...
}
```

main.c

```
#include “linked_list.h”
extern int num_elements;

int main(void)
{
   num_elements = 0;
   ...
   num_elements++;
```



OR …

linked_list.h

```
struct Student {
   char student_id[ID_SIZE];
   ...
   struct Student *next, *prev;
};
extern int num_elements;
void insert(struct Student **list_head,
   struct Student *element);
void delete(struct Student **list_head,
   char *id);
```

main.c

```
#include “linked_list.h”


int main(void)
{
   num_elements = 0;
   ...
   num_elements++;
```

linked_list.c

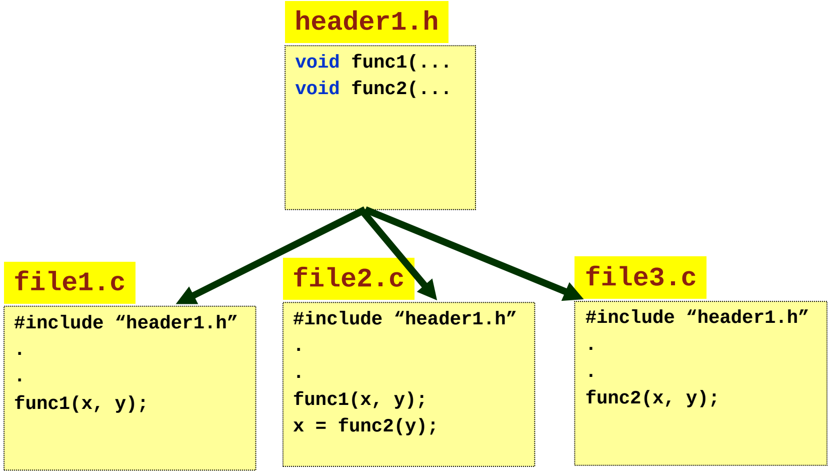
```
#include “linked_list.h”

int num_elements;

/* insert a new node in a linked list */
void insert(struct Student **list_head,
    struct Student *element)
{
  ...
}

/* delete a node from a linked list */
void delete(struct Student **list_head,
    char *id) {
  ...
}
```

extern

```
extern int j;
```

- A keyword of the C programming language
- Informs the compiler that the variable **j** is defined in another file (different source file).
- Compiler does not allocate space when it encounters **extern** declarations.
- Reference to the variable **j** is resolved at the link time.

Using Header Files



header1.h

```
void func1(...
void func2(...
```

file1.c

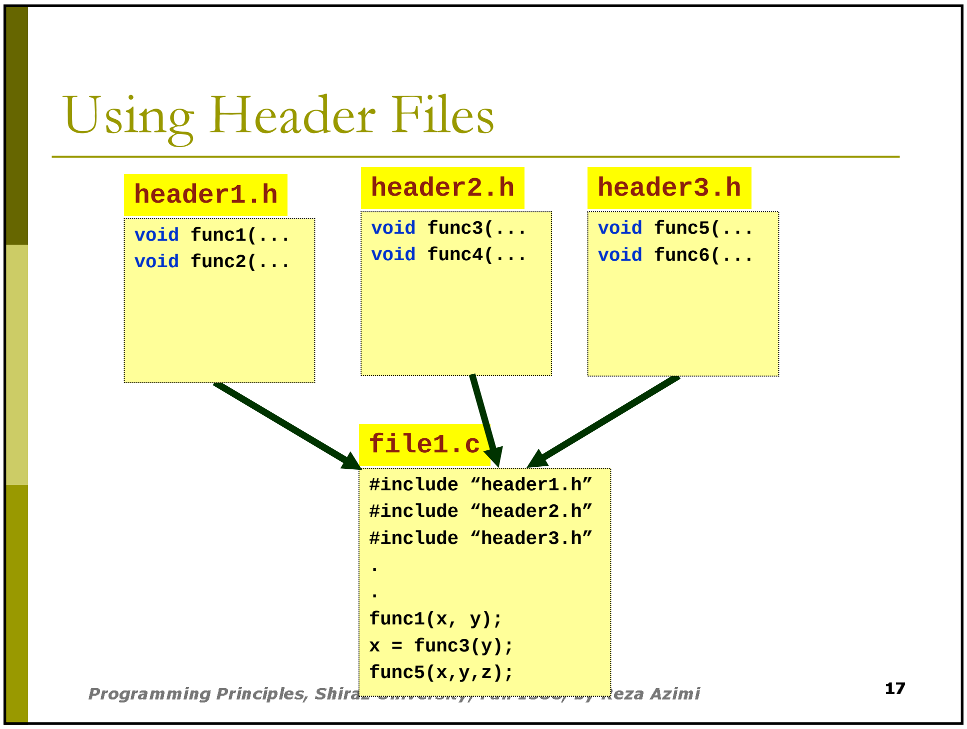
```
#include "header1.h"
.
.
func1(x, y);
```

file2.c

```
#include "header1.h"
.
.
func1(x, y);
x = func2(y);
```

file3.c

```
#include "header1.h"
.
.
func2(x, y);
```



Using Header Files

header1.h

```
void func1(...
void func2(...
```

header2.h

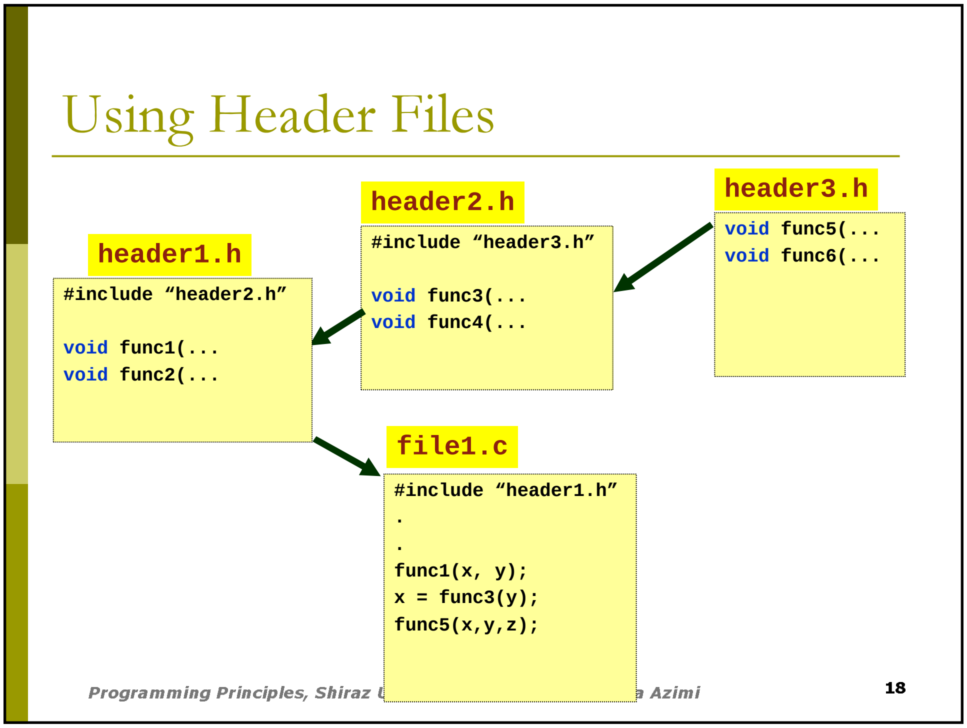
```
void func3(...
void func4(...
```

header3.h

```
void func5(...
void func6(...
```

file1.c

```
#include “header1.h”
#include “header2.h”
#include “header3.h”
.
.
func1(x, y);
x = func3(y);
func5(x,y,z);
```



Using Header Files

header1.h

```
#include “header2.h”

void func1(...
void func2(...
```

header2.h

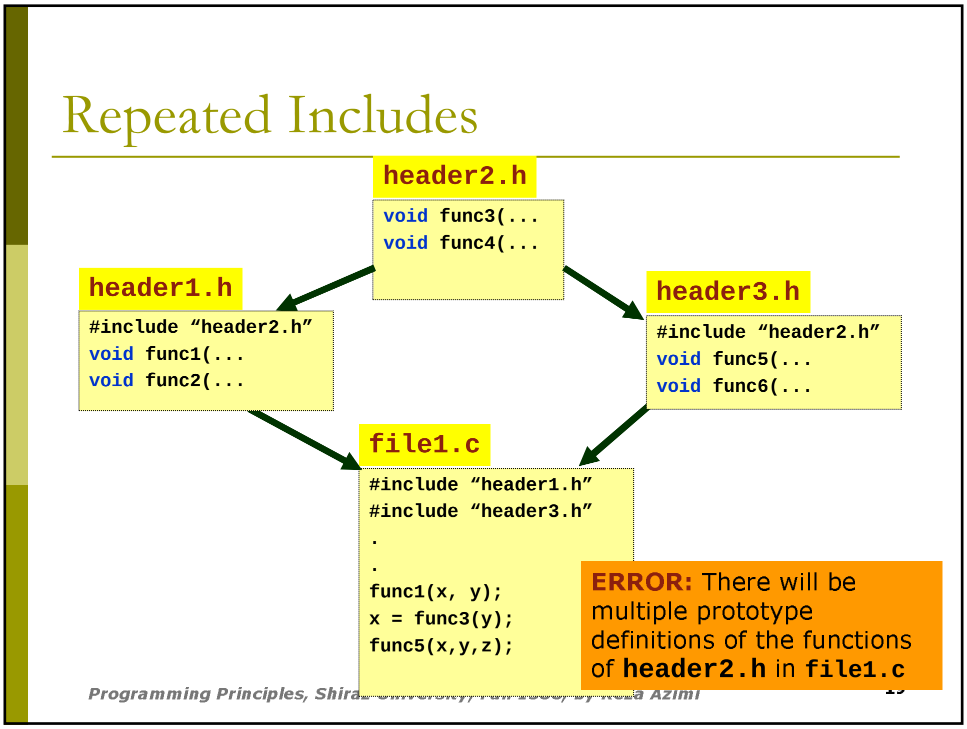
```
#include “header3.h”

void func3(...
void func4(...
```

header3.h

```
void func5(...
void func6(...
```

file1.c

```
#include “header1.h”
.
.
func1(x, y);
x = func3(y);
func5(x,y,z);
```



Repeated Includes

header2.h

```
void func3(...
void func4(...
```

header1.h

```
#include "header2.h"
void func1(...
void func2(...
```

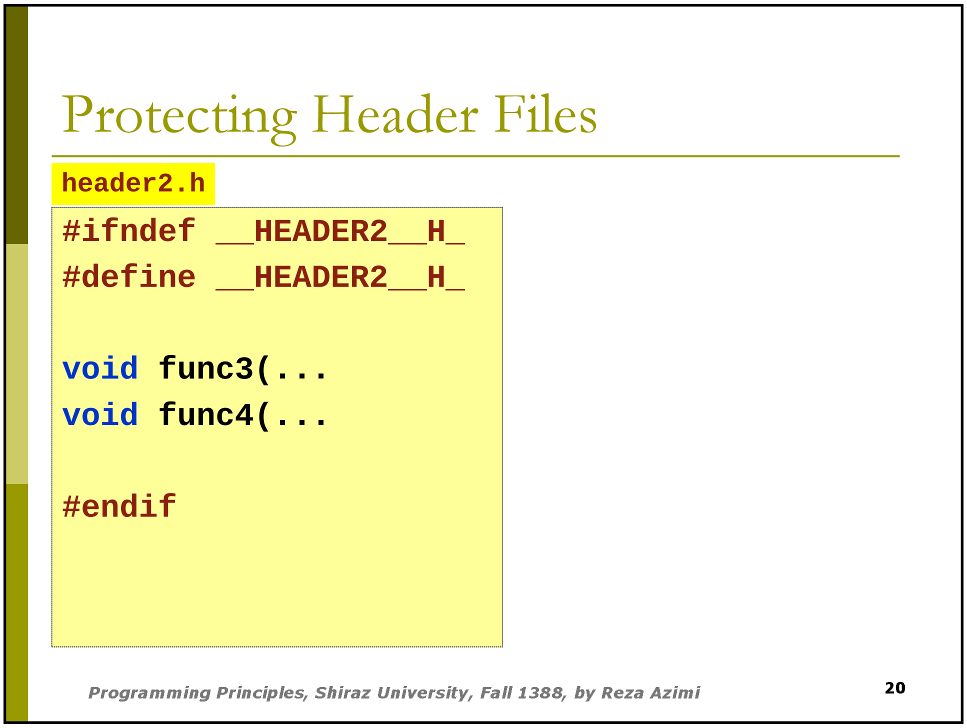
header3.h

```
#include "header2.h"
void func5(...
void func6(...
```

file1.c

```
#include "header1.h"
#include "header3.h"
.
.
func1(x, y);
x = func3(y);
func5(x,y,z);
```

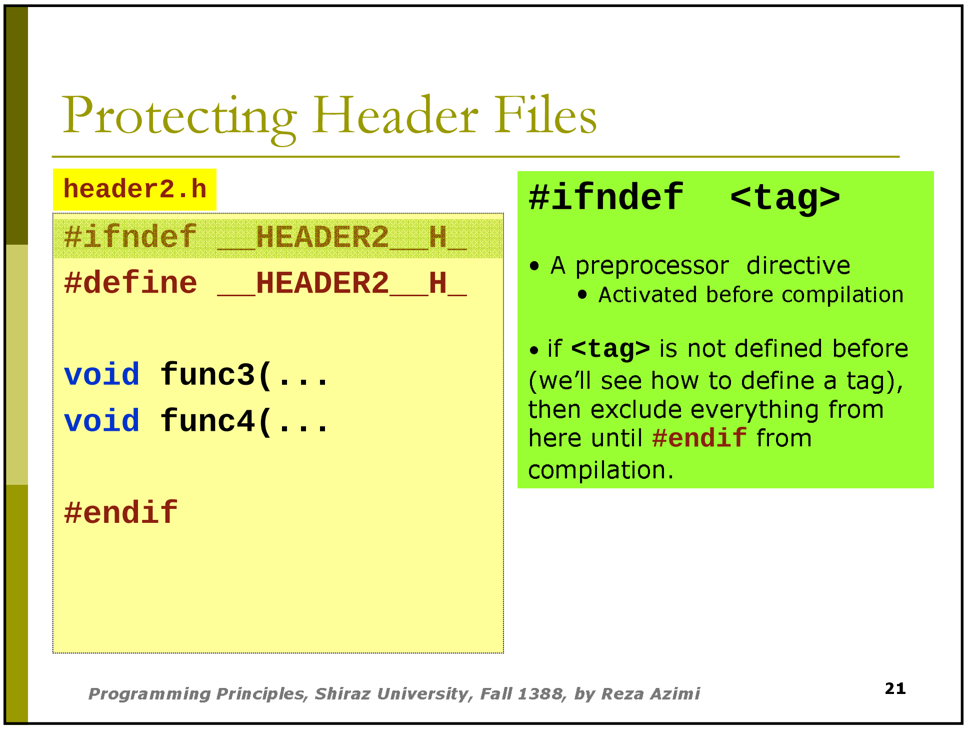
ERROR: There will be multiple prototype definitions of the functions of **header2.h** in **file1.c**



Protecting Header Files

header2.h

```
#ifndef __HEADER2__H_
#define __HEADER2__H_

void func3(...
void func4(...

#endif
```



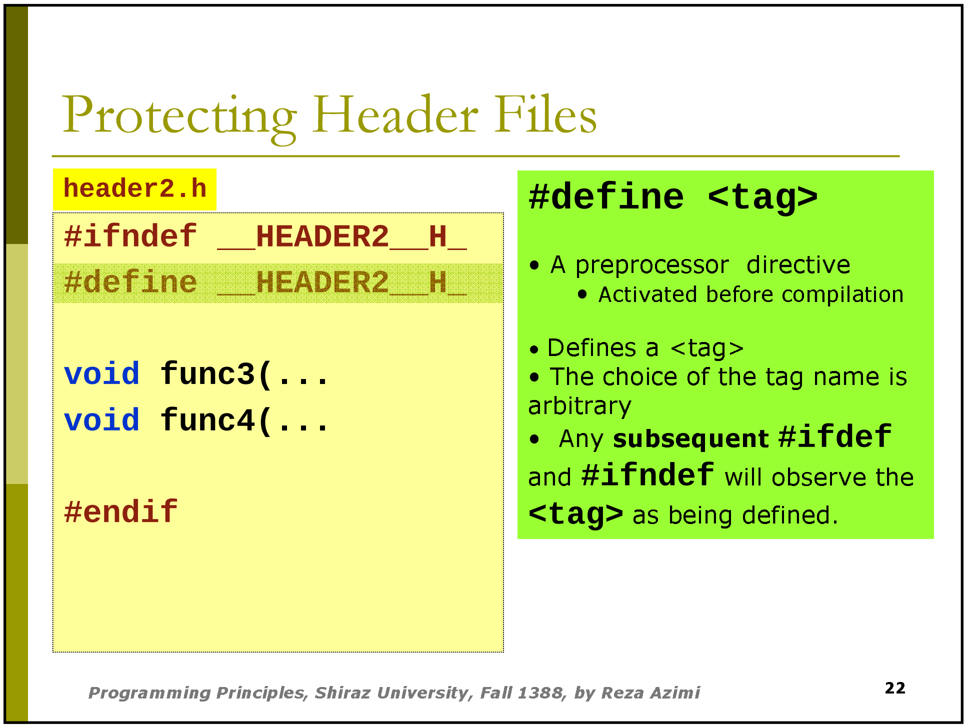
Protecting Header Files

header2.h

```
#ifndef __HEADER2__H_
#define __HEADER2__H_

void func3(...
void func4(...

#endif
```

#ifndef <tag>

- A preprocessor directive
  - Activated before compilation
- if **<tag>** is not defined before (we'll see how to define a tag), then exclude everything from here until **#endif** from compilation.



Protecting Header Files

header2.h

```
#ifndef __HEADER2__H_
#define __HEADER2__H_

void func3(...
void func4(...

#endif
```

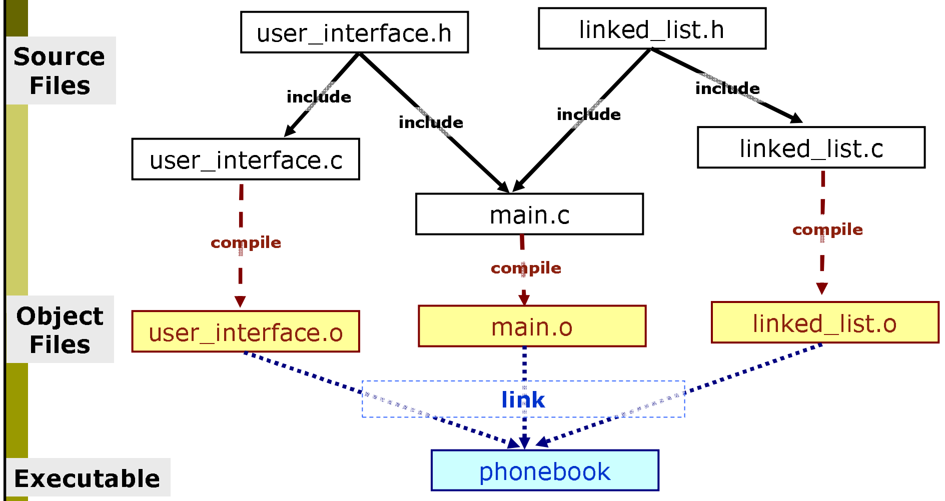
#define <tag>

- A preprocessor directive
  - Activated before compilation
- Defines a <tag>
- The choice of the tag name is arbitrary
- Any **subsequent #ifdef** and **#ifndef** will observe the **<tag>** as being defined.

“Building” Programs

- Building Means:
  - Preprocessing
  - Compiling
  - Linking
- Key Questions
  - If a program consists of many files, does it matter which file is compiled first?
  - If we modify a file, which files need to be recompiled.

Example: Phonebook



Source Files

user_interface.h

linked_list.h

include

include

include

include

user_interface.c

main.c

linked_list.c

compile

compile

compile

Object Files

user_interface.o

main.o

linked_list.o

link

Executable

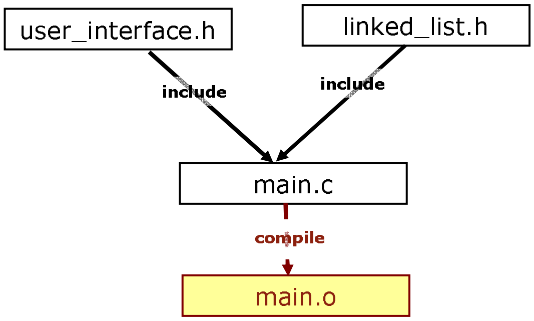
phonebook

Build Steps

```
% gcc -c linked_list.c
% gcc -c user_interface.c
% gcc -c main.c

% gcc -o phonebook main.o
  linked_list.o user_interface.o

% ./phonebook
```

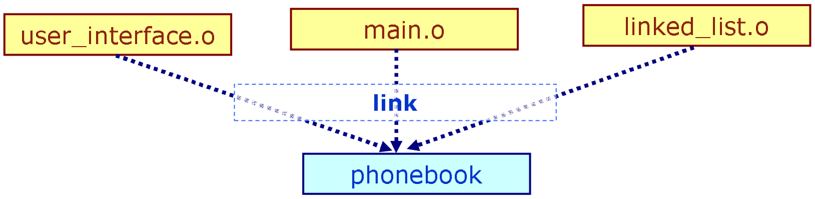
Dependencies (وابستگی)

- Object To Source Files:
  - If either the source or the header file changes, the object file needs to be regenerated.



Dependencies (وابستگی)

- Executable to Object Files:
  - If any of the object files that are linked to produce the executable changes, the link step must be done again.

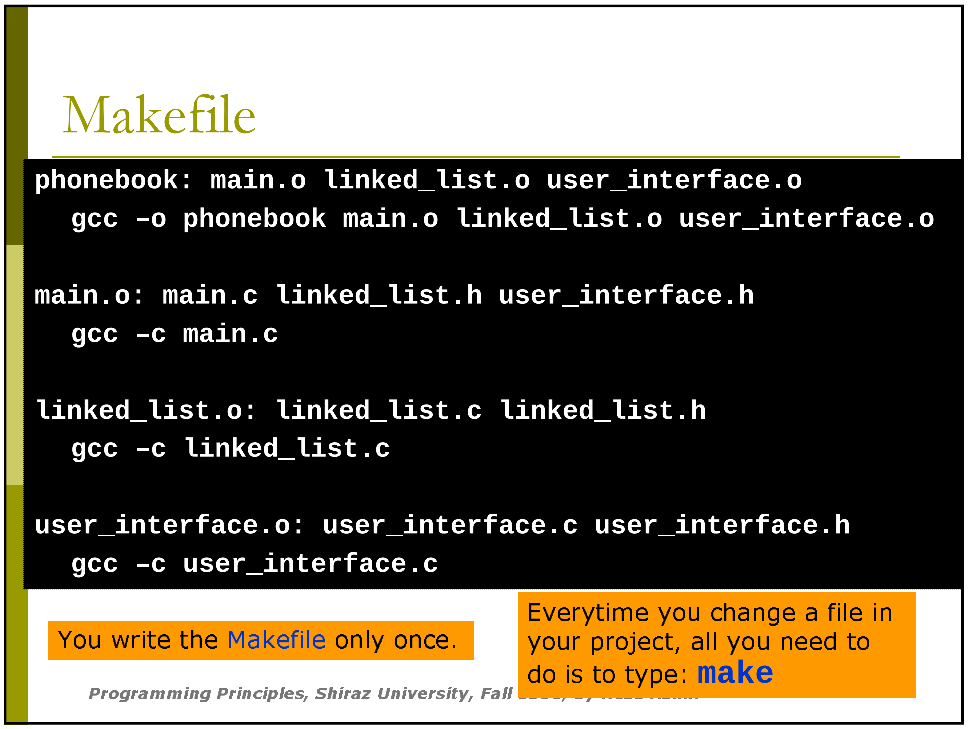


`make` Utility

- A tool to automate the build process

- Programmer needs to write a **`Makefile`** (**`makefile`**) to describe
  - Dependencies
  - Commands for Updating (Recreating) Files

- The **`make`** command reads the **`Makefile`** and builds the dependence graph and builds the executable

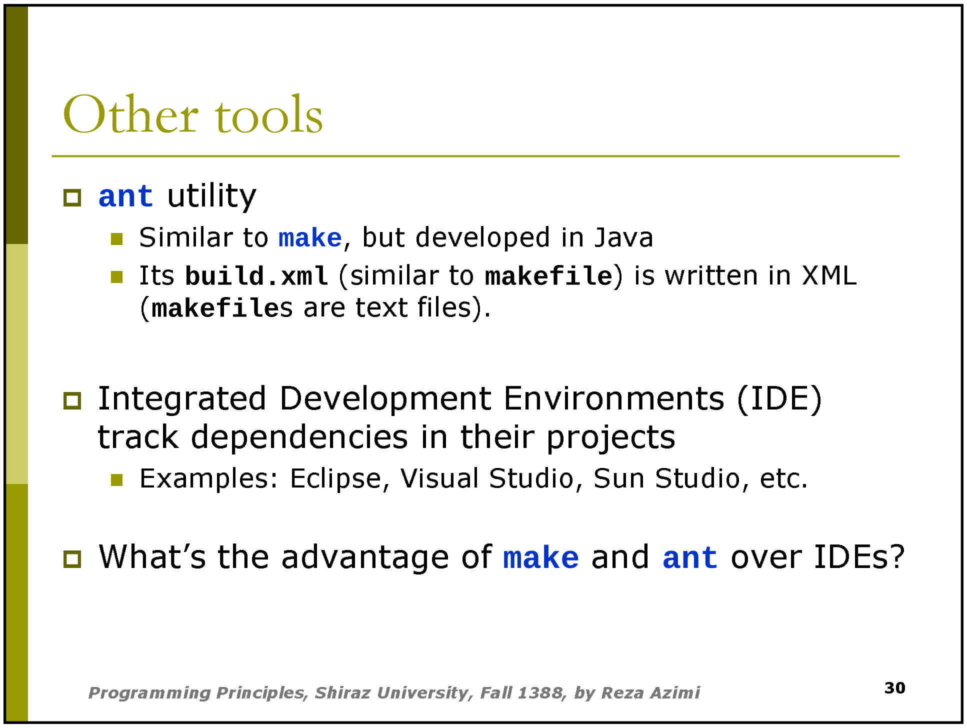


Makefile

```
phonebook: main.o linked_list.o user_interface.o
  gcc -o phonebook main.o linked_list.o user_interface.o

main.o: main.c linked_list.h user_interface.h
  gcc -c main.c

linked_list.o: linked_list.c linked_list.h
  gcc -c linked_list.c

user_interface.o: user_interface.c user_interface.h
  gcc -c user_interface.c
```

You write the Makefile only once.

Everytime you change a file in your project, all you need to do is to type: **make**



Other tools

- **ant** utility
  - Similar to **make**, but developed in Java
  - Its **build.xml** (similar to **makefile**) is written in XML (**makefile**s are text files).
- Integrated Development Environments (IDE) track dependencies in their projects
  - Examples: Eclipse, Visual Studio, Sun Studio, etc.
- What’s the advantage of **make** and **ant** over IDEs?