CSCI 2132
Software Development

Lecture 27:

Compiling and Debugging Large Programs

Instructor: Vlado Keselj

Faculty of Computer Science

Dalhousie University

Previous Lecture

- Command-line arguments
- sortwords.c example with insertion sort
- Writing large programs: header files and "`.c`" files
- modules and header files
- Protecting header files from double-inclusion
- prj-dec2bin and stack example (started)

Dividing a Program into Files

- Example: `prj-dec2bin`
- Step 1: Breaking program logically into source files (`*.c`)
  - `dec2bin.c`: the main program
  - `stack.c`: Stack implementation

Step 2: Sharing

- Sharing type definitions
  - In `bit.h`: `typedef int Bit;`
- Sharing macro definitions
  - Not needed as `STACK_SIZE` is used by `stack.c` only
- Sharing function prototypes
  - `stack.h`
- Advantages of using both `bit.h` and `stack.h` instead of one header file:
  - `bit.h` could be used by another program

Protecting Header Files

- Issue: nested header files
- Example:

```
In stack.h:              | In stack.c:
...                      | ...
#include "bit.h"         | #include "bit.h"
...                      | #include "stack.h"
                         | ...
```

- A problem is that `bit.h` will be included twice in `stack.c`

Conditional Compilation

- Example (`bit.h`):

```
#ifndef BIT_H
#define BIT_H
typedef int Bit;
#endif
```

- Meaning:
  - If `BIT_H` is not defined:
    - Define `BIT_H`
    - Include other code up to:
  - `#endif`

The Final Project Files

- Finally, we are going to break up the original program `dec2bin.c` into the following files:
- `bit.h`: Bit header file, defining the type `Bit`
- `stack.h`: Stack header file, or stack interface
- `stack.c`: Stack implementation
- `dec2bin.c`: the main program

bit.h:

```
/* File: bit.h */
#ifndef BIT_H
#define BIT_H

typedef int Bit;

#endif
```

```
/* File: stack.h */
#ifndef STACK_H
#define STACK_H

#include <stdbool.h>

#include "bit.h"

void make_empty();
bool is_empty();
bool is_full();
void push(Bit i);
Bit pop();
void stack_overflow();
void stack_underflow();

#endif
```

```
/* File: stack.c */
#include <stdio.h>
#include <stdlib.h>

#include "bit.h"
#include "stack.h"

#define STACK_SIZE 100

Bit contents[STACK_SIZE];
int top = 0;

void make_empty() {
  top = 0;
}

bool is_empty() {
  return top == 0;
```

```
}

bool is_full() {
  return top == STACK_SIZE;
}

void push(Bit i) {
  if (is_full())
    stack_overflow();
  else
    contents[top++] = i;
}

Bit pop() {
  if (is_empty())
    stack_underflow();
  else
    return contents[--top];
```

```
}

void stack_overflow() {
  printf("Error: stack overflow!\n");
  exit(EXIT_FAILURE);
}

void stack_underflow() {
  printf("Error: stack underflow!\n");
  exit(EXIT_FAILURE);
}
```

```
/* Program: dec2bin.c */
#include <stdio.h>

#include "bit.h"
#include "stack.h"

int main(void) {
  int decimal;
  Bit bit;

  printf("Enter a decimal integer: ");
  scanf("%d", &decimal);

  while (decimal > 0) {
    bit = decimal % 2;
    push(bit);
    decimal /= 2;
  }
```

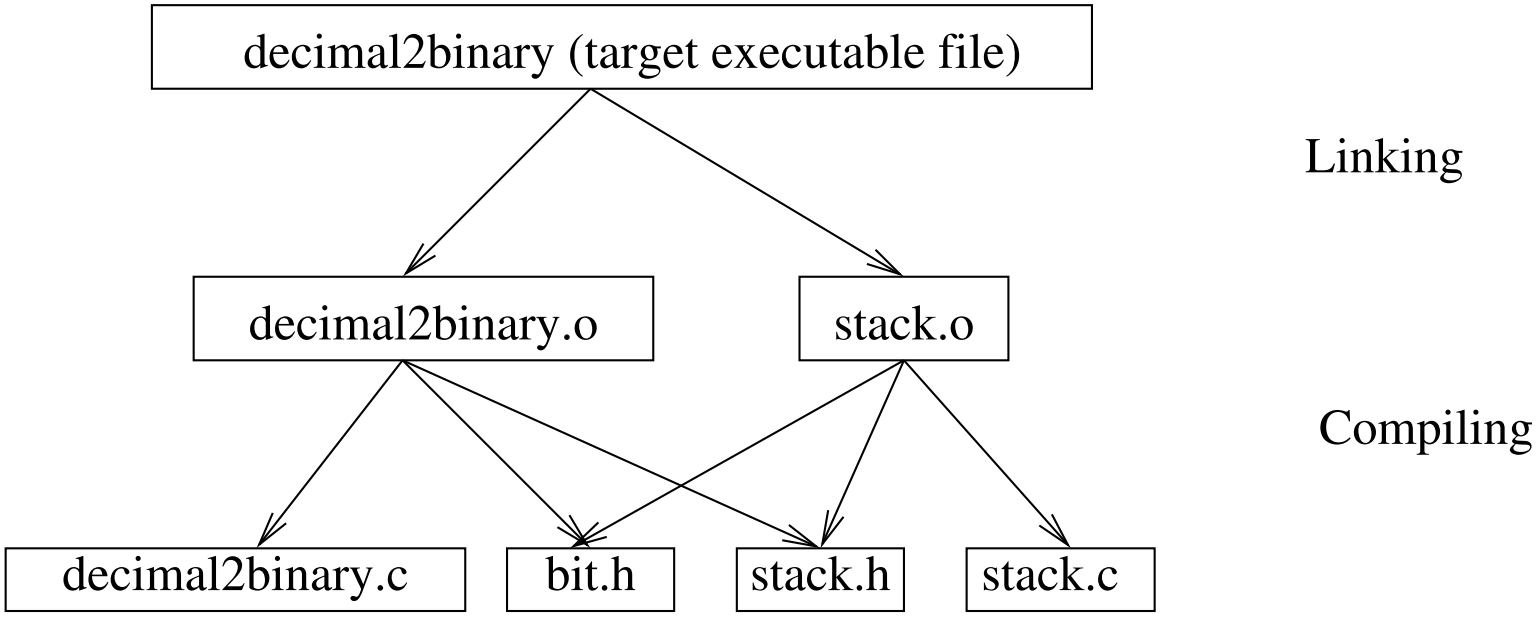
```
    printf("This number can be expressed in binary as: ");

    while (!is_empty()) {
      printf("%d", pop());
    }

    printf("\n");

    return 0;
}
```

The Make Utility

- How do we compile these files?
- The make utility
  - Manages the compilation and linking of multi-file software
  - Reads a makefile (named `makefile` or `Makefile`) that specifies:
    - The targets to be built
    - Commands used to build them
    - How the modules of a software system depend on each other

Dependencies

- A directed acyclic graph (DAG)
- Object file (`*.o`): a file containing machine instructions of one module
- We typically generate one object file for each `*.c` file



Make Dependencies

- If a dependency file has changed more recently that a target, the target is rebuilt
  - Particularly useful for a large project with many files, where not everything needs to be rebuilt
- General syntax:

```
TARGET : DEPENDENCIES
        COMMAND
        COMMAND
        ...
```

- Describes a general recipe how to build a target

Make Recipe Examples

- target: `hello` dependency: `hello.c`

```
hello: hello.c
        gcc -o hello hello.c
```

- More examples:

```
dec2bin.o: dec2bin.c stack.h bit.h
        gcc -std=c99 -c dec2bin.c
dec2bin: dec2bin.o stack.o
        gcc -std=c99 -o dec2bin dec2bin.o \
          stack.o

all: dec2bin hello
```

More about Makefile Syntax

- Important to use Tab character; e.g., in

```
hello: hello.c
	gcc -o hello hello.c
```

- it must be a Tab character before gcc:

```
hello: hello.c
tab gcc -o hello hello.c
```

- Another way which `make` allows:

```
hello: hello.c; gcc -o hello hello.c
```

- Use of `gcc` option: `-c` (compilation without linking)

A Full Makefile Example

```
# makefile for the project dec2bin
.PHONY: help
help:
        @echo 'make all              will produce all'
        @echo 'make dec2bin will produce only '\
                                        'dec2bin'
        @echo 'make clean            will clean'

all: dec2bin hello

dec2bin: dec2bin.o stack.o
        gcc -std=c99 -o dec2bin \
          dec2bin.o stack.o

dec2bin.o: dec2bin.c stack.h bit.h
        gcc -std=c99 -c dec2bin.c
```

```
stack.o: stack.c stack.h bit.h
        gcc -std=c99 -c stack.c

hello: hello.c
        gcc -o hello hello.c

clean:
        rm dec2bin dec2bin.o stack.o \
           hello
```

Using `make` from Command Line

- `make`
  - Makes the first target
  - In our example target ‘help’ which prints makefile help
- `make` *target*; for example:
  - `make dec2bin`
  - `make all`
  - `make clean`

Using `gdb` for Large Programs

- In Makefile use `-g` for all `gcc` commands
- Use `break filename:line_number` to set a breakpoint
- Use `break filename:function_name` (`filename` can be omitted)

Structures

- **Structure:** collection of data items of various types
  - Array: collection of data items of the same type
- Structure elements called members
- Members referred by name
- Some important uses of structures:
  - Data records
  - Building blocks of data structures