Faculty of Computer Science, Dalhousie University *7-Nov-2018*

CSCI 2132 — Software Development

Lecture 27: Compiling and Debugging Large Programs

Location: Chemistry 125 Instructor: Vlado Keselj
Time: 12:35 – 13:25

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`

If one header file includes another header file, there are chances that we include one header file twice in a program. In our example, the header file `stack.h` includes the file `bit.h`, so if we include both header files `bit.h` and `stack.h` in a program file, as we do in `stack.c` the contents of the file `bit.h` will be included two times.

This double-inclusion of one header file is not necessarily an issue, and it may go without errors through compilation. However, if we have type definitions for example (`typedef`), the compiler will not allow it, since we will have type definitions of the same types repeated. Another good reason to avoid double inclusions is that we could create circular inclusions which cannot be processed by the preprocessor.

The solution is to protect each header file from double-inclusion using preprocessor conditional compilation.

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;
}
```

21.1 Compilation of Large Programs

We break a large C program, such as our program `decimal2binary` in several files in order to be able to better manage the source code. As programs get large, there are many reasons why we want to break them into several files, such as a better ability to reuse code by using only some files from one program in the other program, or for better collaboration of several programmers on the same project by having each of them working only on a set of files assigned to them.

Compilation of large C programs can take a long time and this leads to another reason for breaking large C program into several files: after changing the source code we do not necessarily need to recompile all files. We have seem so far that several C source files can be compiled using one compiler command, such as:
`gcc -o dec2bin dec2bin.c stack.c`
which will compile both files `dec2bin.c` and `stack.c` into one executable program named `dec2bin`.

We can also separately compile individual C files and then link them into the final executable program. Individual compilation is achieved using the option `-c` of the compiler `gcc`. For example, the command:
`gcc -c dec2bin.c`
will produce the file `dec2bin.o`, which is called the *object* file, and it contains machine code, or object code, generated from the source file `dec2bin.c`. While this object code contains machine code, it is not ready for execution yet and cannot be run as a command. Similarly, using the command:
`gcc -c stack.c`

we can generate the file `stack.o`, which contains object code generated from the file `stack.c`. The two object files can be used to create the final executable `dec2bin` using the command:
`gcc -o dec2bin dec2bin.o stack.o`

Even though the last command is very similar to the command:
`gcc -o dec2bin dec2bin.c stack.c`
the command that uses only `.o` files is faster because it will only do the linking operation among object files in order to create executable, while the command with `.c` files will first compile all C files into object files on the fly, and then link them into the executable file.

21.2 The Make Utility

Slide notes:

The Make Utility

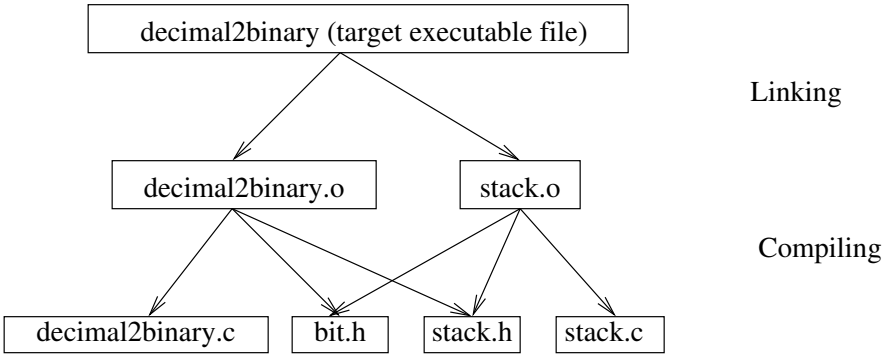
- How do we compile these files?
- The make utility
  - Manages the compilation and linking of multi-file software
  - Reads a makefile (named `makefile` or `Makefile`) that specifies:
    - The targets to be built
    - Commands used to build them
    - How the modules of a software system depend on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

Since we know how to compile source code files into individual object code files, we can sometimes save file on compiling a program. Using the previous example, if we modify the file `stack.c` we can simply ran the command `gcc -c stack.c` to make `stack.o` file and then only like files into the final executable `dec2bin` and avoid compiling the `dec2bin.c` file. Keeping track of which files have been changed, which files need to be compiled again or linked again, and what are exact commands to do these operations is time-consuming and error-prone if done manually, so we use a utility, or tool, called 'make' to do this automatically.

The processes of compiling a file into an object file, and linking object files into an executable file can be regarded as a more general process of generating a file depending on other files. The make utility relies on us to record which files are generated from which other files and what are the commands used to invoke the generation process.

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

The behaviour of the make tool is defined by the 'makefile', which is a set mostly a set of rules about how different files are built from other files. The general form of a rule in a makefile is:

```
target: dependencies....
        recipe..
```

The *target* is the file name of the file created by the rule. The *dependencies* are the source files; i.e., the files which when updated will trigger recreation of the target file. The *recipe* is one or more commands that are executed in orther to produce the target file. The recipe follows the line with target and dependencies. It can consist of several lines, which must be indented starting with a tab character.

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)

```
# 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)

22 Structures

Similarly to arrays being collection of data items of the same type, a *structure* in C is a collection of data item of optionally various types, called structure *members.*

While in arrays we refer to each element by an index, in a structure we refer to each element by a member name.

When comparing to object-oriented programming, such as C++ and Java, the structures were predecessors of objects, and structure types correspond to classes.

Structures have various uses, such as modeling natural data records, such as student records, which may contain data of different types, such as names, IDs, and birth dates. They are also important building blocks in many data structures, such as linked lists and trees.