ECEN 449 – Microprocessor System Design

Some tips to good programming and Makefiles

Objectives of this Lecture Unit

- Some tips to good programming

Variable names

- Use descriptive variable names
  - int i ; versus int student_exam1_score;
  - Even though longer names may take more space, much better later for understanding the code
- Write Comments
  - Much easier for later review
  - Others may have to pick up your code
- Use braces to delineate loops --easier to visualize
- for (i=0; i<100; i++) a[i] = i;
- for (i=0; i<100; i++)

```
{
  a[i] = i;
}
```

Input to a function/program

- Always check the input to a function whether returned from another function or user input
- If input is expected to be an integer between 1 and 10
  - input_to_my_function = fn();
  - if ((input_to_my_function >10) || (input_to_my_function <1))

```
    printf ("Invalid input received\n");
else
  {
    /* do my work here */
  }
```

Input to functions

- More important when dealing with devices
- Many possible error conditions
- Need to deal with all cases
- Not sufficient to just deal with expected or correct input
- if (return_code == error_code)

```
{
  /* do error handling */
}
```

- Much of the code may deal with errors –ok.

Arithmetic operators

- salary_per_month = total_salary/working_months;
- What is the problem?
- working_months = working_days/30;
- radius = sqrt(area/pi); --??
- scanf(“%d\n”, &year);
  - my_function(year)

```
{
  /* what should we be careful about? */
}
```

Style

- Good style keeps problems away
- Have a header file for each function prototype
- **********
- my_function: produces enlightenment
- input: hours (in integers ranging from 1 to 2)
- output: smart students (in integers with IQ from 120 to 140)
- error: negative IQs
- Dependencies: depends on functions ECEN350 ENGR112
- **********

Passing values/pointers

- Pass information as much as possible with values
- Less problematic
- When using pointers, always cast a pointer to what you expect it to be
  - int_pointer = (int *) intp;

Makefiles

- Why are they useful?
- How do they work?

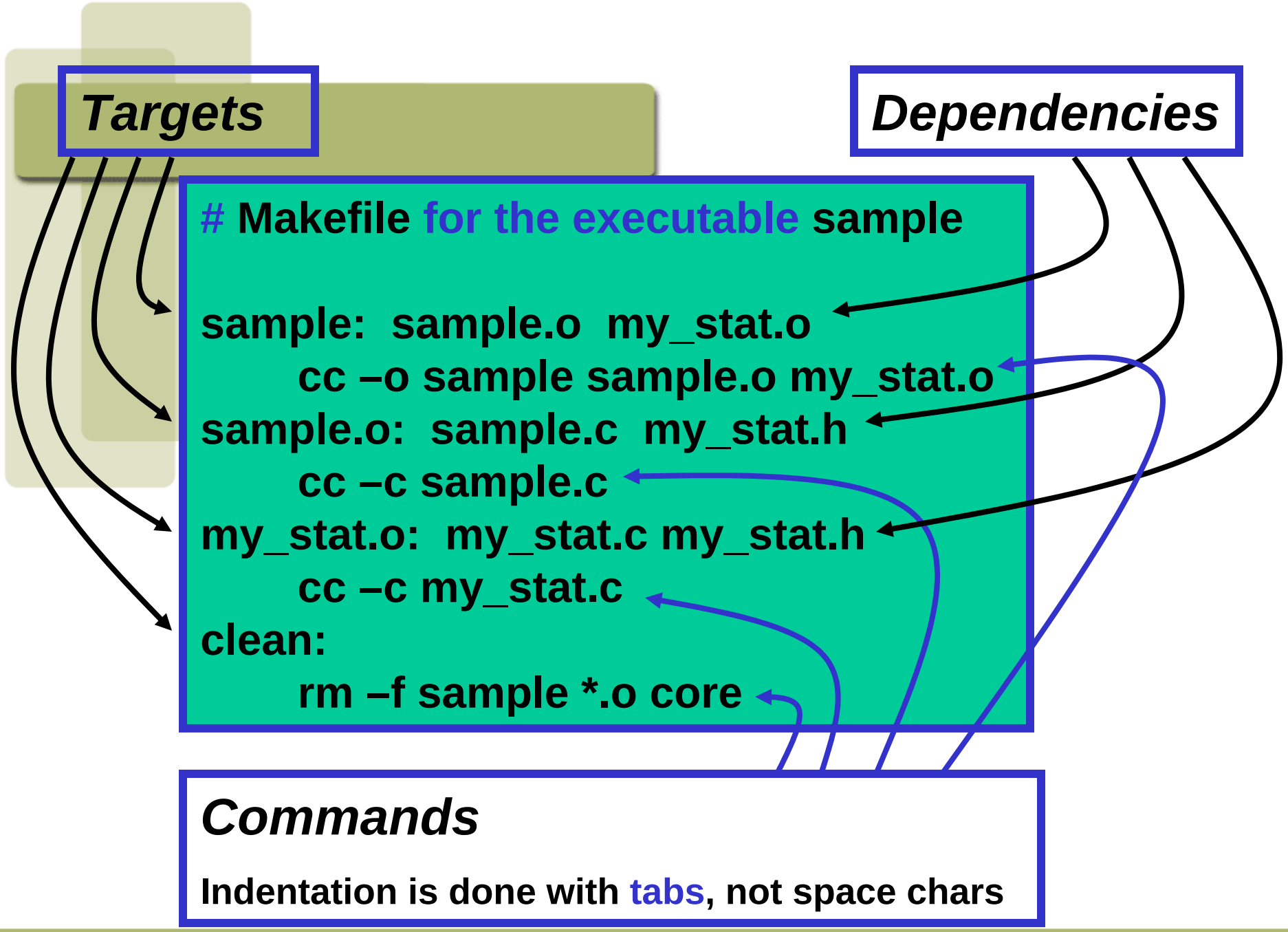
» This portion of this lecture unit is adapted from the notes of a UWO course

Multiple Source Files

- Large C programs typically use many files
- A large program is divided into several sections, possibly in different C files.
- Each C file is compiled separately to produce a ".o" file
- Object files are then are linked to generate the executable for the program
- C programs are generally broken up into two types of files
  - .c files:
    - Contain source code and global variable declarations
    - Compiled once and never included
  - ***.h*** files:
    - Used for module interfaces (function prototypes), type and **struct** definitions, **const** and **#define** constant declarations

Why do we need the "make" utility

- Programs consisting of many C-files are nearly impossible to maintain manually
- The **make** utility can be used to:
  - Ensure that only those C-files that were modified since the last compilation, are recompiled before a new executable is built
  - Automate the compilation process


Targets
Dependencies
Makefile for the executable sample
sample: sample.o my_stat.o
cc –o sample sample.o my_stat.o
sample.o: sample.c my_stat.h
cc –c sample.c
my_stat.o: my_stat.c my_stat.h
cc –c my_stat.c
clean:
rm –f sample *.o core
Commands
Indentation is done with tabs, not space chars

How to use the **"make"** utility

- Save the file with name **Makefile** (or **makefile**) in the same directory as the ***.c*** and ***.h*** files
- Every time you want to build/rebuild your target program, type **make**
- The **make** utility will
  - Find the **Makefile**
  - Check rules and dependencies to see if any C-files need recompilation
  - Regenerate the necessary files that need updating, ensuring that the target is up to date
  - Example: if only **my_stat.c** is modified since the last build, then **my_stat.o** and **sample** will be created.

Targets of the **"make"** utility

- To regenerate sample.o only, without creating a new executable:

```
make sample.o
```

- To remove all generated files (executable file, object files, core files):

```
make clean
```

- To force all files to be recompiled, use:

```
make clean;  make
```

Default target

- If a target is not specified when make is invoked, Unix assumes that the ***first*** target in **makefile** is to be generated; otherwise, it generates the target that has been specified
- **make** uses file time stamps (time of last modification) to decide whether to rebuild a target
- To force the time stamp on a file to be updated without really editing it, use the unix command “touch”.

```
touch my_stat.h
```

Using Macros in **Makefiles**

- Macros can be used in **Makefiles** to reduce file size by providing (shorter) names for long or repeated sequences of text
- ***Example***: The definition **name = text string** creates a macro called **name** whose value is **text string**
- Subsequent references to **$(name)** or **${name}** are replaced by **text string** when the **Makefile** is processed
- Macros make it easier to change **Makefiles** without introducing inconsistencies

Using Macros

```
CC = gcc
HDIR = ../Include
INCPATH = -I$(HDIR)
DEPH =  $(HDIR)/StackTypes.h  $(HDIR)/StackApi.h
SOURCE = StackApi
export:  $(SOURCE).o
$(SOURCE).o:  $(SOURCE).c  $(DEPH)
        $(CC)  $(INCPATH)  -c  $(SOURCE).c
print:
        lpr  $(SOURCE).c
clean:
        rm  -f  *.o
```