CSI 402 – Lecture 1

(Programs in Multiple Files and make)

Program in a Single File

File: prog.c

```
#include <stdio.h>
int  main(void) {
  int   max(int, int);  float average(int, int);
  int i = 17, j = -20;
  printf("%d\n", max(i, j));  printf("%f\n", average(i, j));
  return 0;
}
int max (int a, int b) {
   if (a >= b)
      return a;
   else return b;
}
float average (int a, int b) {
   return (a+b)/2.0;
}
```

Program in a Single File (continued)

Unix command to generate `a.out`:

```
        gcc   prog.c
```

Difficulties:

- If the source file is large, it is harder to fix the syntax errors.
- Even if all the errors are in just one function, the whole file must be recompiled.

Solution: Split program into several small files.

Program in Multiple Files: Example 1

File: `main.c`

```
#include <stdio.h>
int  main(void) {
  int   max(int, int);  float average(int, int);
  int i = 17, j = -20;
  printf("%d\n", max(i, j));  printf("%f\n", average(i, j));
  return 0;
}
```

Example 1 (continued)

File: max.c

```
int max (int a, int b) {
   if (a >= b)
      return a;
   else return b;
}
```

File: avg.c

```
float average (int a, int b) {
   return (a+b)/2.0;
}
```

Unix command to generate a.out:

```
     gcc   main.c   max.c   avg.c
```

Example 1 (continued)

A Better Method: Compile the files separately and link them to get the executable version.

```
gcc  -c  main.c
gcc  -c  max.c
gcc  -c  avg.c
gcc   main.o  max.o  avg.o
```

Remarks:

- The "-c" option specifies "compile only".
- Note the use of gcc for linking the object files.
- Additional information is given in Handout 1.1.
- Using make, the above process can be considerably simplified (to be discussed later in this lecture).

Program in Multiple Files: Example 2

Note: Here, variables are shared across files.

File: `main.c`

```
#include <stdio.h>
int   x, y;
float z[10];    /* x,y,z : global. */

int  main(void) {
  void   xy_change(int);
  void   z_change(int, float);

  x = 15;  y = 17; z[3] = 5.2;
  xy_change(3);  printf("%d  %d\n", x, y);
  z_change(3, 7.4);  printf("%f\n", z[3]);
  return 0;
}
```

Example 2 (continued)

File: `funct.c`

```
/* x, y, z : externally defined. */
/* Size not specified for z. */

extern  int    x, y;
extern  float  z[];

void xy_change (int a) {
    x += a;  y -= a;
}
void z_change (int a, float x) {
    z[a] = x;
}
```

Common practice: Header files (files with extension “`.h`”).

Example 2 with Header Files

File: `globals.h`

```
int     x, y;
float   z[10];
```

File: `externs.h`

```
extern  int     x, y;
extern  float   z[];
```

Example 2 with Header Files (continued)

File: `main.c`

```
#include  <stdio.h>
#include  "globals.h"

int  main(void) {
  void   xy_change(int);
  void   z_change(int, float);

  x = 15;  y = 17; z[3] = 5.2;

  xy_change(3);  printf("%d  %d\n", x, y);
  z_change(3, 7.4);  printf("%f\n", z[3]);

  return 0;
}
```

Example 2 with Header Files (continued)

File: `funct.c`

```
#include  "externs.h"

void xy_change (int a) {
    x += a;  y -= a;
}
void z_change (int a, float x) {
    z[a] = x;
}
```

To produce a.out:

```
   gcc  -c  main.c
   gcc  -c  funct.c
   gcc   main.o   funct.o
```

Note: Header files are not specified in the compile command.

Other Header Files

(a) Symbolic Constants:

File: `constants.h`

```
#define  MINKEY    1
#define  MAXKEY    100
```

Note: File `constants.h` can be included in other source files.

Other Header Files (continued)

(b) Structure Definitions:

File: `struct_def.h`

```
struct  key_record {
   int  value;  struct key_record *next;
};
typedef  struct key_record*  keyptr;
```

Note: File `struct_def.h` can also be included in other source files.

Other Header Files (continued)

(c) Function Prototypes:

File: `prototypes.h`

```
void   insert_key(int);
void   print_list(void);
```

Note: File `prototypes.h` can also be included in other source files.

A complete example: Handout 1.2.

Summary Regarding Header Files

Typical Header Files:

- `constants.h`
- `struct_def.h`
- `globals.h`
- `externs.h`
- `prototypes.h`

Remarks:

- Header files `constants.h`, `struct_def.h` and `prototypes.h` can be included in any source file.

Summary Regarding Header Files (continued)

Remarks (continued):

- Header file `globals.h` is typically included in the source file containing `main`.
- Header file `externs.h` is typically included in source files containing functions that access the global variables.
- The `extern` attribute applies only to **variables**; it cannot be used for constants or structure definitions.

Introduction to make

Basic Information Regarding make:

- make: A Unix tool for software development.
- Generates commands to separately compile and produce executables.

Example: Suppose a C program consists of two source files main.c and funct.c.

Normal Unix command to generate executable file prog:

```
gcc   main.c  funct.c  -o  prog
```

Note: This becomes tedious when there are many C source files.

A Simple `makefile`

File: `makefile`

```
prog:    main.o  funct.o
         gcc  main.o funct.o -o prog
main.o:  main.c
         gcc  -c  main.c
funct.o: funct.c
         gcc  -c  funct.c
```

Important Note

Each gcc command line above starts with the tab character.

Unix command to generate `prog`:

```
make
```

A simple makefile (continued)

Remarks:

- prog : Default target.
- main.o, funct.o : Other targets.
- For each target, the makefile specifies
  - dependency information and
  - the Unix command needed to generate the target. (Each command line starts with the "tab" character.)
- The make Unix command
  - looks for a file named makefile (or Makefile) in the current directory and
  - tries to create the default target.
- To create main.o, the Unix command is:

```
make   main.o
```

Additional Examples for `make`

Example 1: Handout 1.3.

Example 2: Handout 1.4.

More Information: A. Oram and S. Talbott, “Managing Projects with `make`”, O’Reilly & Associates, Inc., 1996. (ISBN: 0-937175-90-0)