Lecture 3
C Programming Language

Summary of Lecture 3

- streams
- error handling
- library functions
- break, continue, switch statements
- constants and macros
- C preprocessor (CPP)
- header files

Streams

- Stdin, stdout and sdterr are buffer I/O streams.
- Other I/O streams can be defined, e.g.
  FILE * fp;
- Since the type FILE is defined in the standard I/O library, we need to include the line:
  #include <stdio.h> in the source code.
- To open/create a new stream, we use fopen with the following prototype:

  FILE *fopen(char * filename, char *mode);
  mode can be “r” for read, “w” for write.
- We use the functions fprintf and fscanf that work like printf and scanf but get the stream pointer as argument:
  fprintf(fp,”this file’s name is %s”,filename);
- <u>IMPORTANT:</u> close the stream (file) after use:
  fclose(FILE *fp);
  Example: fclose(fp);

Stderr

- Stderr is another example of a stream.
- Just like stdin and stdout, it is predefined and does not have to be explicitly opened by the programmer.
- Stderr is used for error messages. These messages are displayed on the screen by default, even when we redirect the program's output (e.g. `a.out > output_file`).
- Redirect error messages by:
  `a.out 2> error_file`
- Example:

```
if ( (fp=fopen("p.txt","r")) == NULL)
{
      fprintf(stderr,"Cannot open file\n");
      exit(-1);
}
```

Library Functions

- Library functions are commonly needed functions that have been predefined.
- C has several standard library functions
- To use a library function, include the appropriate header file and link in the library during compilation.
  Example: #include <math.h> in “p1.c”
  and then % gcc p1.c -lm

- Not including these files can lead to potential problems:
  Unless we add #include <math.h> the output
  to this program:

```
main()
{
    printf("2 cubed is %f\n",pow(2,3));
}
```

  is 0.000

Standard Libraries

- Math #include <math.h>
- String #include <string.h>
- Input/Output #include <stdio.h>
- Dynamic Memory Allocation
  #include <stdlib.h>

Constants and Macros

- #define <ident> <token-sequence>
  #define <ident>(<params>) < token-sequence>
- Syntax: no “=“ before (<params>)
- Macros are expanded by the C preprocessor
  (e.g. every appearance of <ident> is replaced by <token-sequence>
- Use:

```
#define MAX_STR_LEN  20
#define IS_UPPER(c)   ((c)>=‘A’ && (c)<=‘Z’)
#define IS_LOWER(c)   ((c)>=‘a’ && (c)<=‘z’)

char arr[MAX_STR_LEN+1], *str;
….
for (str=arr;*str != ‘\0’; str++)
{
      *str = TO_LOWER(*str);
}
```

Macros Pitfalls

- #define SQR(x) x*x
- Operator Precedence Errors:

  ```
  SQR(a + b);
  ```

  is expanded to: `a + b*a + b`

  and **not**: `(a + b)*(a + b)`

  <u>Solution :</u>

  Put parentheses (or braces) around Macro

  ```
  #define SQR(x) ((x) * (x))
  ```
- Side Effects Errors:

  ```
  SQR(i++);
  ```

  expanded to: `i++ * i++`

  which increments i twice.
- Unnecessary Function Calls:

  ```
  SQR(long_function(a,b,c));
  ```

  will evaluate the function twice.
- There are no general solutions for the last two errors - so be cautious and wise !

Calling a Function vs. Calling a Macro

Calling a Function	Calling a Macro
Always an expression	May be a **statement** (require automatic variables)
Will not change arguments, no side effects	May have side-effects
Can always carry a newly created object	May require an argument to carry a newly created object
Limited to **fixed type arguments**	Operates (usually) on arguments of varying types
Saves executable code	Code is duplicated
May be passed as an argument to other functions	**Cannot** be passed as an argument
Function call overhead (for stack handling)	No calling overhead

When is a Macro better than a Function ?

- Rules of Thumb:
  - operation required is **short, simple** and (maybe) used in **different locations** (files).
  - operation required is **short, simple** and is used **intensively**.
  - operation required is performed on variety of **different types**.
-
- Examples of last case:

```
#define MAX(a,b)    (((a)>(b)) ? (a) : (b))
#define SWAP(type,a,b)
                {type t=a; (a)=(b); (b)=t;}
```

<u>Note:</u>

The expression `(cond) ? stmt1 : stmt2 ;`

is a shortcut for:

```
if (cond)   stmt1
else   stmt2
```

Enumerable Types

- Types that consist of certain **integral values** and are carried by **symbolic names**
- Enum definitions:
  ```
  enum bool {FALSE,TRUE};
  enum month
  {JAN=1,FEB=2,…,DEC=12};
  enum colors
  {WHITE=1,BLACK,GREEN=8,RED};
  ```
- Using enum types
  ```
  enum bool b[10];
  enum cond = FALSE;
  ```
- enum vs. #define (enum is superior)
  - The compiler may check for type mismatch.
  - the debugger may recognize the symbolic names.

Switch Statement

- switch (month) {

```
        case JAN: /* stmt */
        case FEB: /* stmt */
                        …
        case DEC:   printf("31 days\n");
                    break;
        case APR:
                    …
        case NOV:   printf("30 days\n");
                    break;
        case FEB:   if (leap_year)
                        printf("29 days\n");
                    else
                        printf("28 days\n");
                    break;
        default:            printf("month
error\n");
                    break;
}
```

Break

- Can also be used in for, while, do-while loops
- break terminates these loops early, control
  transfers to the first stmt after the loop
- Example:

```
/* this function returns 1 if "a" is in
   increasing order,  0 otherwise */
int monotonic(int a[], int N)
{
      int i;
      for (i=0; i<=N-1;i++)
      {
         if (a[i+1] < a[i])
              break;
      }
      if (i==N) return 1;
      else return 0;
}
```

Continue

- The continue statement transfers control to
  the next iteration of the loop.
- Example:

```
char s1[12] = “Donald Duck”;
char s2[12] = “Jerald Burk”;
char m[12];
int i,count = 0;
for (i=0;i<=12;i++)
{
     if (s1[i] != s2[i])
        continue;
     m[count++] = s1[i];
}
printf(“%s\n”,m);

/* m = ? */
```

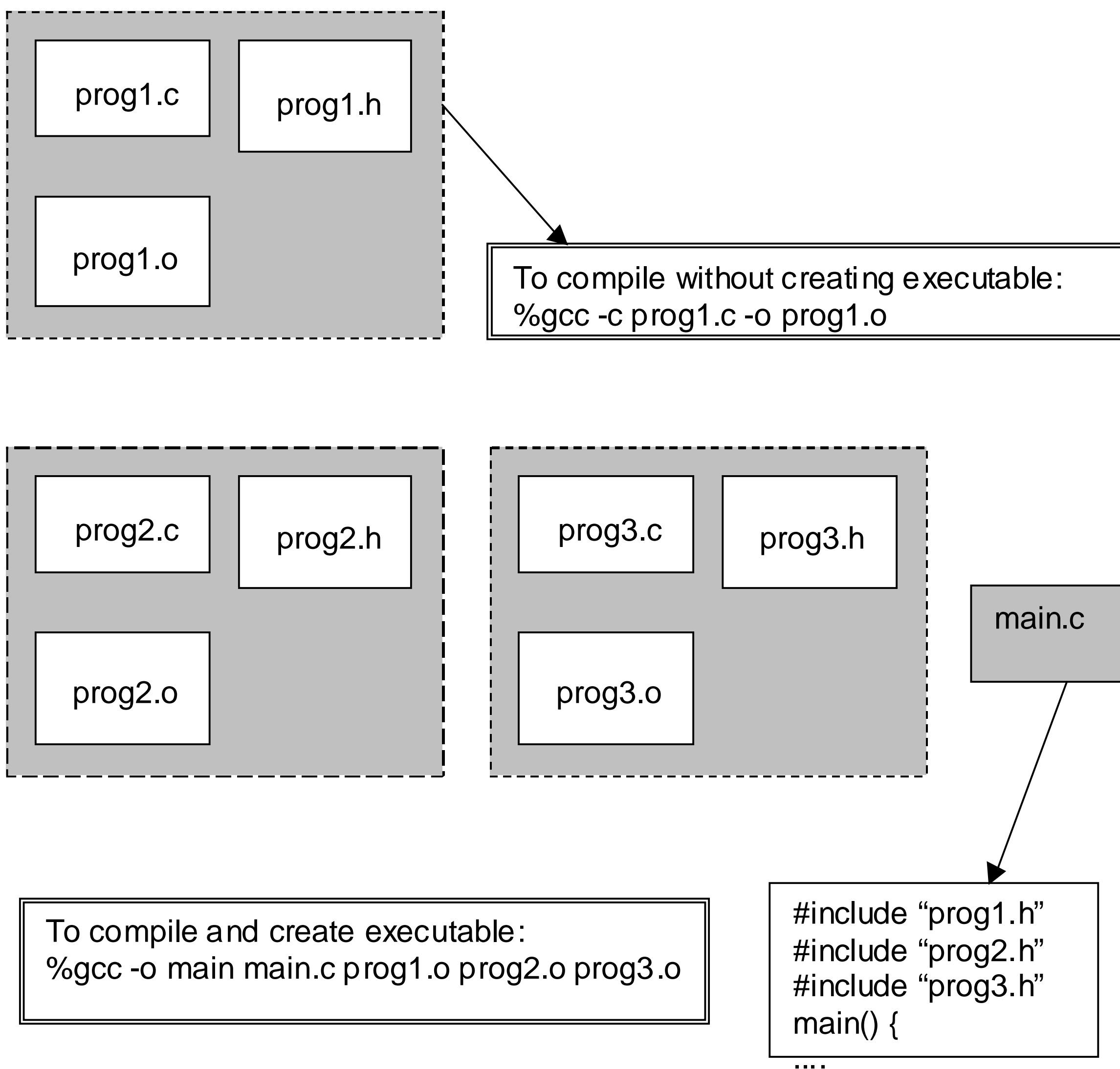
C Preprocessor

- Program goes through CPP before other compilation
- CPP operations:
  - automatic (e.g. deleting comments)
  - requested (CPP directives)
- CPP Directives:
  #include - include header files
  #define - define constants or macros
  #ifndef, #endif - conditional inclusion

Header Files

- What are header files for?
- Simple interface to previously defined functions (contain only declaration)
- Modularity: code up small components, each with different functionality, and then link them together
- Each component has a `.c` file and a `.h` file
- The `.c` file has the function definitions. The `.h` file has the function prototypes, constants definitions, macro definitions.
- Easier to debug and to reuse

From Source to Executable



Conditional text inclusions

- A common use of #ifndef is in header files
- It is usually harmful to include a header file more then once (p1.c includes h1.h that includes h2.h)
- The way to prevent this is inserting this macro at the beginning of every header file you write:

```
#ifndef _header_name
#define _header_name
/* prevents entering here in future
inclusions*/
…
```

  and then insert this line at the end of file:

```
#endif
```