Text Files
C Preprocessor

CS2023 Winter 2004

Outcomes: Text Files and C Preprocessor

- “C for Java Programmers”, Chapters 5 & 6
- After the conclusion of this section you should be able to
  - Write programs that perform I/O on text files
  - Describe how the C preprocessor works
  - Use preprocessor directives to:
    - Define macros with and without parameters
    - Include system and user-defined header file
    - Enable conditional compilation

File Handles and Opening Files

```
FILE *fileHandle;
```

```
fileHandle = fopen(fileName, fileMode);
```

Examples

```
FILE *f;
FILE *g;
```

```
f = fopen("test.dat", "r");
g = fopen("test.out", "wb");
```

Opening Files

```
fileHandle = fopen(fileName, fileMode);
```

`"r"`	open for input; (file must exist)
`"w"`	open for output; (overwrite or create)
`"a"`	open for output; (always append to this file)
`"r+"`	like `"r"` for I/O (file must exist)
`"w+"`	like `"w"` for I/O (overwrite or create)
`"a+"`	like `"a"` for I/O

The above modes may be used to specify a *binary* mode, by using the character **`b`**

Closing Files and Predefined Handles

```
fclose(fileHandle);
```

File handles are *resources* that you have to manage:

close files as soon as you do not need them!

You can use three predefined file handles in your programs:

`stdin`	the standard input stream
`stdout`	the standard output stream
`stderr`	the standard error stream

File Handling Idioms

```
if((fileHandle = fopen(fname, fmode)) == NULL)
      /* failed */

if(fclose(fileHandle) == EOF)
       /* failed */
```

Basic File I/O Operations

- `int getchar()` `int fgetc(fileHandle)`
- `int putchar(int)` `int fputc(int, fileHandle)`
- `int scanf(…)` `int fscanf(fileHandle, …)`
- `int printf(…)` `int fprintf(fileHandle, …)`

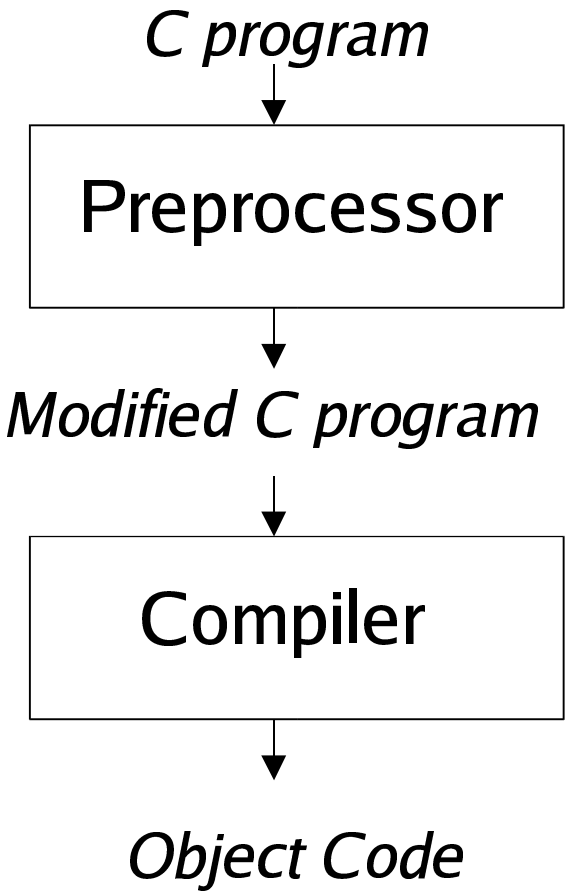
```
/*
 * Program that reads three real values from the
 * file "test" and displays on the screen the
 * sum of these values
 */
int main() {
   FILE *f;
   double x, y, z;

   if((f = fopen("test", "r")) == NULL) {
      fprintf(stderr, " can't read %s\n", "test");
      return EXIT_FAILURE;
   }
```

```
    if(fscanf(f, "%lf%lf%lf", &x, &y, &z) != 3) {
        fprintf(stderr, "File read failed\n");
        return EXIT_FAILURE;
    }

    printf("%f\n", x + y + z);

    if(fclose(f) == EOF) {
        fprintf(stderr, "File close failed\n");
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}
```

Idiom!

The Preprocessor

- macros (with and without parameters)
  - `#define`
- conditional compilation
  - `#ifdef`
- file inclusion
  - `#include`


C program
Preprocessor
Modified C program
Compiler
Object Code

Input to Preprocessor

```
/* Converts a Fahrenheit temperature to Celcius */

#include<stdio.h>

#define FREEZING_PT 32.0
#define SCALE_FACTOR (5.0 / 9.0)

int main()
{
 float fahrenheit, celcius;

 printf("Enter Fahrenheit temperature: ");
 scanf("%f", &fahrenheit);

 celcius = (fahrenheit - FREEZING_PT) * SCALE_FACTOR;

 printf("Celcius equivalent is: %.1f\n", celcius);

 return 0;
}
```

Output from Preprocessor

```
< lines brought in from stdio.h>



int main()
{
 float fahrenheit, celcius;

printf("Enter Fahrenheit temperature: ");
 scanf("%f", &fahrenheit);

 celcius = (fahrenheit - 32.0) * (5.0 / 9.0);

 printf("Celcius equivalent is: %.1f\n", celcius);

 return 0;
}
```

Paramaterless Macros

```
#define  macroName  macroValue
```

During preprocessing, each occurrence of **macroName** in the source file will be replaced with the text specified in **macroValue**.

```
#define PI              3.14
#define SCREEN_W          80
#define SCREEN_H          25
#define PROMPT     Enter two \
                integers
```

Preprocessing Guidelines

- Macros names will always appear in upper case.
- Any constant value, which might change during software development should be defined as a macro, or as a constant.
- By using macros, you are adding new constructs and new functionality to the language – if you do this inappropriately, the readability of your code may suffer.

Macros with Parameters

`#define macroName(`*`parameters`*`) macroValue`

Examples

```
#define RANGE(i)  (1 <= (i) && (i) <=
  maxUsed)

#define IS_EVEN(n)  ((n)%2==0)
```

Parenthesize aggressively!

Why so many ()'s?

```
#define TWO_Pi 2*3.14159
    conversion_factor = 360/TWOPI;
```

Becomes

```
    conversion_factor = 360/2*3.14159;
#define SCALE(x) (x*10)
    j = SCALE(i+1);
```

Becomes

```
    j = (i+1*10);
```

Macros with Parameters

- Advantage:
  - Program may be faster
  - Programs more generic

```
#define MAX(x,y) ((x)>(y)?(x):(y))
```

  can be used to find the larger of two types of integers, floats, …
- Disadvantages
  - No type checking
  - Can make programs harder to read

Macro errors

```
#define PI = 3.14

#define PI 3.14;

#define F (x) (2*x)
```

File Inclusion

`#include "filename"` the *current* directory and ...

`#include <filename>` special *system* directories

All relevant definitions may be grouped in a single **header** file

screen.h:

```
#define SCREEN_W    80
#define SCREEN_H    25
```

```
#include "screen.h"
int main() {
…
}
```

Conditional Compilation

- Good for debugging:

```
#define DEBUG
…
#ifdef DEBUG
printf("Value of i: %d\n", i);
printf("Value of j: %d\n", j);
#endif
```

- Can also set a macro at compile time:
  - `gcc -DDEBUG prog.c`

Header files

To avoid multiple inclusion:

```
#ifndef SCREEN_H
#define SCREEN_H
...
/* contents of the header */
#endif
```