COMPSCI 1XC3 - Computer Science Practice and Experience: Development Basics

# Topic 13 - The C Preprocessor and Miscellaneous Errata

NCC Moore

McMaster University

Summer 2021

Adapted from Chapters 13 and 14 of C: How to Program 8th ed., Deitel & Deitel

Preprocessor Directives

Signal Handling

## So what's left?

Not much! We need to go over some of the advanced preprocessor commands:

- `#define`
- `#undef`
- `#if` and `#endif`
- `#error` and `#pragma`
- `#line`

The C preprocessor is responsible for handling all of the operations that need to occur before a source file can be tokenized, an abstract syntax tree produced, and an executable generated.

- Comments must be removed
- Preprocessor directives must be processed.
  - A preprocessor directive is some manipulation of the text of the source file.

## Macro, Macro Man! (Reprisal)

At the beginning of the course we explained how macros can set symbolic constants, but we can do more with them than just specifying the size of an array!

```
#define <identifier> <replacement text>
```

- Remember that a macro is a *character substitution operation*
- This means that a macro can be used to insert *anything* into a C source file. Including:
  - Semicolons and curly braces, complete or partial expressions, assignments, argument lists and function calls
- Macros have no grammatical limitations, though the substitutions they make must result in grammatically correct C code.
- It is recommended to keep these substitutions to one line.
- Macros are named in all-caps by convention.

## Macro, Macro Man! (Reprisal, cont.)

```
// A syntactically correct C program:
#include<stdio.h>

#define BEGIN (){
#define TWICE(x) x x
#define INT_PRINT(x) printf("%d\n", x);
#define AREA(x,y) ((x)*(y))

int main BEGIN
  if (1 == 1) {
    INT_PRINT(AREA(8,10));
TWICE(})
```

```
80
```

## Macros with Arguments!

You may have noticed the program on the previous slide contained macros with a function-like construction.

- Yes! You can make your macros take arguments!
- The process somewhat resembles lambdas in Python:

```
(lambda x y : x*y)(8,10)
```

- The main difference of course is that we are performing *character substitutions only*, so the arguments to a pragma may be *any group of characters*, so long as the operation results in valid C code.
- For example, the macro `TWICE(x) x x` takes the input `x` and inserts two of them into the file.
- This is used to duplicate the closing curly brace, ending both `main` and it's if-statement.

## Undefined Reference

If you want to restrict the applicability of your macro, you can use the following to remove macros:

```
#undef <identifier>
```

- Remember that macros are also imported when you include a library or one of your own source files.
- By using `#undef`, you can prevent macros you define in one file from being applied to another.
- Once a macro has been undefined, it can subsequently be redefined, so it can also be useful if you want a macro to perform different substitutions in different parts of your source file.
  - The above does not seem an advisable use-case, as it seems rather spaghetti-code-like.

## Shampoo and Conditioner

We can hide code from the compiler conditionally using #if and #endif

```
#if <condition>
Code which may be deleted
#endif
```

- Any code between #if and #endif will be deleted if the condition evaluates to 0, and will be kept otherwise.
- The conditional expression must have integral type and can include only integer constants, character constants, and the "defined" operator.
  - Remember! This command is being executed before your program compiles. As such, functions, variables, enumerations, and typedefs should not be used!
  - Best to stick to Macros, integer literals and the standard operators.

## When to be Iffy

Common use-cases of `#if` includes:

- Preventing a header file from being included multiple times in the same project.
- Giving a program a debug mode.
  - Typically, in debug mode a program will execute many printf() statements which reveal internal data, or generate a log file.
- Setting different constants / executing different statements, depending on system-defined values.
  - This could be the word size of the processor, the version number of the operating system, or any number of things.
  - This assumes the program will be compiled from source on multiple different systems.

## If Variants

Here are some quality of life improvements!

- The `defined(<Macro>)` operator returns 1 if the specified macro is defined, 0 otherwise.
- `#elif` and `#else` may be used in the same way as `else if` and `if` to create additional branches.
- `#ifdef x` is shorthand for `#if defined(x)`
- `#ifndef x` is shorthand for `#if !defined(x)`

You can also use `#if` to comment out blocks of code.

- Using `/*` and `*/` doesn't allow for nesting, but `#if` and `#endif` do!

## Miscellaneous Directives

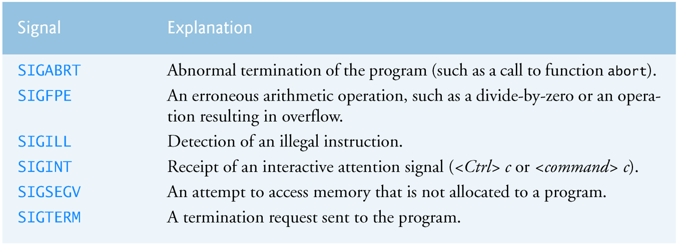
- `#error <tokens>` prints an error message
  - What the error message says depends on your system and C implementation
  - Halts compilation upon execution.
- `#pragma <tokens>` performs miscellaneous commands
  - Which pragmas are available depends on your system and C implementation.
  - Using pragmas may make your code less portable!
- `#line <integer>` manually changes the line number of the source file.
  - Generally used to generate more meaningful compiler warnings and errors.

## Sending out Signals

C DOES NOT SUPPORT EXCEPTION HANDLING

- It handles **signals** instead, which, as with most things in C, are not quite the same thing.
- For each process, the operating system maintains 2 integers which indicate the status of various signals.
  - One is for pending signals
  - The other blocks pending signals.
- The signal handling library of C is `<signal.h>`
- This library contains a number of defined values, which indicate various signal types.
- Most of these signals correspond closely to the types of exceptions you saw in python.

## Signal Types



| Signal | Explanation |
|---|---|
| `SIGABRT` | Abnormal termination of the program (such as a call to function `abort`). |
| `SIGFPE` | An erroneous arithmetic operation, such as a divide-by-zero or an operation resulting in overflow. |
| `SIGILL` | Detection of an illegal instruction. |
| `SIGINT` | Receipt of an interactive attention signal (<*Ctrl*> *c* or <*command*> *c*). |
| `SIGSEGV` | An attempt to access memory that is not allocated to a program. |
| `SIGTERM` | A termination request sent to the program. |

**Fig. 14.5** | `signal.h` standard signals.

## Not Handling Not Exceptions

- In order to handle a signal, you must register a function to activate for an incoming signal of a particular type.

```
signal(<signal>, <function pointer>)
```

- The `signal()` function is provided in `<signal.h>`.
- The function associates an incoming signal type with a function.
- If the signal is received during program execution, execution is paused, and the function is immediately entered.
- Once the function is terminated, the association between signal and function is broken.
  - Fortunately, it is possible to call `signal()` from within the signal-handling function.

## A Somewhat Long but Humorous Example

```c
#include<stdio.h>
#include<signal.h>
#include <unistd.h>

int finish = 0;

void handle_sigint (int signal_Value) {
  finish = 1;
}

int main() {
  signal(SIGINT, handle_sigint);
  printf("Let me tell you a joke...\n");
    sleep(1);
// ...
```

## A Somewhat Long but Humorous Example (cont.)

```
//...
    while (1)
    {
      if (finish) {
        printf("  A: Knock Knock\n"); sleep(1);
          printf("  B: Who's there?\n"); sleep(1);
          printf("  A: Orange.\n"); sleep(1);
          printf("  B: Orange who?\n"); sleep(1);
          printf("  A: Orange you glad I didn't say
   Banana?\n"); sleep(1);
          break;
      } else {
          printf("  A: Knock Knock\n"); sleep(1);
          printf("  B: Who's there?\n"); sleep(1);
          printf("  A: Banana.\n"); sleep(1);
          printf("  B: Banana who?\n"); sleep(1);
//...
```

## A Somewhat Long but Humorous Example (cont.)

```
/...
        printf("   A:  Ba-na\n"); sleep(1);
            printf("          na\n"); sleep(1);
            printf("          na\n"); sleep(1);
            printf("          na\n"); sleep(1);
            printf("          na-na\n"); sleep(1);
          }
      }
    return 0;
}
```

Cue Demo!

Thus concludes our discussion of the C programming language.

CONGRATULATIONS
YOU MAGNIFICENT
HUMAN BEINGS

Students: Now we know everything there is to know about C!
The Prof: