Applied Programming

Using Macros Effectively

Applied Programming

- Parameterized Macros
- Macros and Conditional Compilation
- Debugging Macros

Weird Facts

All C compilers include a preprocessor “**cpp**” that doesn’t know anything about the C language !!

https://www.pinterest.com/Tannia71/strange-but-true

The C Preprocessor

- Always runs as part of the compile and the output is normally invisible

- Lines that begin with a **#** are *preprocessing directives*

- The ***most common*** preprocessing directive are
  - **`#include`** used to include (header) files.
  - **`#define`** used to define macros.

Including Files

- To include **standard libraries** use angled brackets

```
#include <stdio.h>
#include <stdlib.h>
```

  Search for the above files in system directories but not in the current directory!

- To including your custom headers use double quotes

```
#include “myheader.h”
```

  Search for these files **in the current directory** and then in system directories

Include File Hierarchy

- Always include **standard include files FIRST** then personal include files

```
#include <stdio.h>
#include <stdlib.h>
#include “myheader.h”
```

- Personal include files often assume standard include files are present
  - Order often matters
  - Standard first is always safe

Simple Macros

`#define` *identifier token_string*

- Simple macros are traditionally used in C to declare "constants"
  - Traditionally in ALL CAPS
  - Note the use of the "(" and ")", this will be important later

```
#define MAX_ITERATIONS  (1000)
```

- All *macros* and preprocessor directives are *expanded* by the C preprocessor (**`cpp`**) *before the compiler* processes the code
  - Macros are extensively used in C (not so in C++)

Parameterized Macros

- Uses “variables”
- In code will “act” like a function

`#define` *identifier (parameter_list) token_string*

Examples:

```
#define SQ(x)    ((x) * (x))
#define CUBE(x)  (SQ(x) * (x))
```

int I = SQ(2); int j = CUBE(3);

Undefining Macros

- Use **`#undef`** to remove a macro you don't want.
  - Because you want to replace it with something else

```
#undef  identifier
```

Example:

```
#undef SQ
```

Writing “Correct” Macros

- It is *very easy to define wrong macros*

  Quiz: which of the following is correct ?

```
o #define SQ(x) x * x
o #define SQ(x) (x) * (x)
o #define SQ (x) ((x) * (x))
o #define SQ(x)((x) * (x));
o #define SQ(x) ((x) * (x))
```

  (only the last one always works correctly !!)

- Guidelines for “error free” macros
  1) Do not use a semicolon at end of macro (generally true)
  2) Don’t leave space between the macro name and the *parameter_list* parenthesis
  3) Always enclose parameters with parenthesis
  4) Always parenthesize the *token_string*

Macros: Good Practice

Naming Conventions:

- Always **name macros with UPPER CASE** this simplifies debugging

  Warning: there are macros in the standard C library defined with lower case letters ☹

  Illustration: Following this rule a programmer would interpret the following lines of code as follows:

```
value max(a, b);  /* a function call */
value MAX(a, b);  /* a macro         */
```

In this course we will always follow this naming convention

Macros: Good Practice ?

The Standard Libraries:

- Some of the C standard libraries have macros defined with lower case letters ☹

  Example:

  - In **`<stdlib.h>`** we have the *macros*
    `toupper(c)` **`tolower(c)`** **`toascii(c)`**
  - There are also *C functions with exactly the same names !*

Tip: To call the functions write the function names between parenthesis, for example, **`(toupper)(c)`** *calls the function* but **`toupper(c)`** *uses the macro*

Predefined Macros

- In ANSI C there are five predefined macros:

`__FILE__`	Source file name being compiled
`__LINE__`	Current line number in file
`__DATE__`	Date file was compiled
`__TIME__`	Time file was compiled
`__STDC__`	1 if ANSI C, 0 in C++

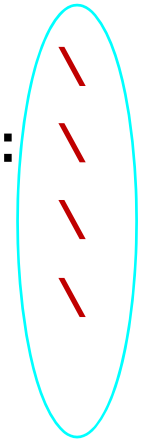
- Predefined macros are named using *two leading and trailing underscore characters* (`__`)

Use __LINE__ and __FILE__ in your error messages

Multi line Macros

Print the values of double variables **`x`**, **`y`** and **`z`** and the name and line of the program where they occur.

```
#define DEBUG_PRINT printf(                    \
          “in File %s at Line %d:              \
          x=%f, y=%f, z=%f”,                   \
          __FILE__,__LINE__,                   \
          x, y, z)
```



Note: *Multiline* macros can be defined by *ending each line*, except the last one, with a *backslash* (****)

Macro Operators # and

- The unary operator ***# converts to a string the formal parameter*** of the macro

  Example:

```
#define PRINT_INT_VAR(var) \
                    printf("%s is %d", #var,var)
```

The expression	expands to
`PRINT_INT_VAR(lcv)`	`printf("%s is %d","lcv",lcv)`

Note that **`#var`** became the string **`"lcv"`**

Macro Operators # and

- The *binary operator* ***## causes concatenation of two formal parameters***

Example

```
#define  X(i)  x ## i
```

The expression	expands to
`X(1) = X(2) = X(3);`	`x1 = x2 = x3;`

Note that **`x ## i`** became **`xi`**

Using ## to create variables

```
#define SAMPLE(R)\
    { int _t_ ## R;                                          \
      for(_t_ ## R=0; _t_ ## R < R; _t_ ## R++) {\
     printf("%d\n", _t_ ## R);                               \
    } /* End for */ } /* End macro */

int main() {
   /* The looping variable will be _t_20  */
   SAMPLE(20);
}
```

Macros outside the box

- Macros don’t have to implement COMPLETE C code and can be used in pairs
  - A macro is really a text substitution language with limited variable support.
  - Normally poor programing style but may be required
    - Document heavily
- When working with macros it can be helpful to just write C code and then “cut and replace” with the macro version

Macros outside the box

- Consider:

```
#define FOR_START(R)    {                    \
                for(i = 0; i < (R); i++) {
#define FOR_END          }}
```

- "C" Code

```
FOR_START(100);
   printf("working");
FOR_END;
```

After macro expansion:

```
{for(i = 0; i < (100); i++) {
  printf("working");
}}
```

- Are there any "hidden" bugs in this code?

Other Directives

Directive	Action
#if MACRO … **#else** … **#endif**	Execute the first enclosed block if MACRO evaluates to a non-zero value, otherwise execute the 2nd enclosed block
#if MACRO1 … **#elif MACRO2** … **#endif**	Execute the first enclosed block if MACRO1 evaluates to a non-zero value, otherwise evaluate MACRO2, etc.
#ifdef MACRO … **#endif**	Execute the enclosed block if MACRO is defined. #if defined MACRO is an alternate form
#ifndef MACRO … **#endif**	Execute the enclosed block if MACRO is NOT defined
#warning "msg"	Causes the preprocessor to print a warning
#error "msg"	Causes the preprocessor to stop and print an error.

Conditional Compilation

- C compilers allow for the definition of simple macro symbols on the command line
  - In **`gcc`** the option **`-D`** has two forms:

`-D`*name* or **`-D`***name=foo*

Example: (you will try this in the homework)

`-DTIMING`	defines the symbol **`TIMING`** with value 1 (*true*)
`-DREPEAT=1000`	defines the symbol **`REPEAT`** with value 1000

- You can undefine these symbols with the directive **`-U`***name* (e.g, **`-UTIMING`**, **`-UREPEAT`**)

Conditional Compilation

Example: Code fragment for testing sparse matrix routines

```
#include <stdio.h>
#include <time.h>
#ifdef SPARSE
  #include “SparseMatrices.h"
#else
  #include “FullMatrices.h"
#endif
#ifdef SPARSE
   printf("Storage for matrices (bytes) = %ld\n",
    (MatrixA.BytesUsed +
   MatrixB.BytesUsed + MatrixC.BytesUsed +
   sizeof(MatrixA) + sizeof(MatrixB) + sizeof(MatrixC)));
   FreeSparseMatrix (&MatrixA);
   FreeSparseMatrix (&MatrixB);
   FreeSparseMatrix (&MatrixC);
#else
   printf("Storage for matrices (bytes) = %ld\n",
      (sizeof(MatrixA) + sizeof(MatrixB) +
    sizeof(MatrixC)));
#endif
```

Macros Subtleties

- Consider a macro to *swap integer values*. The following trick (x-or) is often used (in assembler). Does it work properly ?

```
/* The xor trick */
#define SWAP(a,b) a^=b; b^=a; a^=b;
```

swap_macro1.c

```
int x = 3;
int y = 4;

SWAP(x,y);
if (x<0)
   SWAP(x,y);
```

```
int x = 3;
int y = 4;

x^=y; y^=x; x^=y;;
if (x<0)
   x^=y; y^=x; x^=y;;
```

Because of the semicolon, after expansion only the first statement is under the conditional, the other two always execute

Macros Subtleties

- We could add curly braces

```
/* The xor trick */
#define SWAP(a,b) {a^=b; b^=a; a^=b;}
```

swap_macro2.c

```
int x = 3;
int y = 4;

SWAP(x,y);
if (x<0)
  SWAP(x,y);
```

```
int x = 3;
int y = 4;

{x^=y; y^=x; x^=y;};
if (x<0)
  {x^=y; y^=x; x^=y;};
```

This appears to work fine

Macros Subtleties

- What if we have an else part to the if ?

```
/* The x-or trick */
#define SWAP(a,b) {a^=b; b^=a; a^=b;}
```

swap_macro3.c

```
int x = 3;
int y = 4;
int z = 5;
SWAP(x,y);
if (x<0)
  SWAP(x,y);
else
  SWAP(x,z);
```

```
int x = 3;
int y = 4;
int z = 5;
{x^=y; y^=x; x^=y;};
if (x<0)
  {x^=y; y^=x; x^=y;};
else
  {x^=z; z^=x; x^=z;};
```

Now it does not work (else without if error !)

Macros Subtleties

- We can fix it as follows

swap_macro4.c

```
/* The do while trick */
#define SWAP(a,b) do {a^=b; b^=a; a^=b;} while (0)
                          Now works since SWAP will
                          expand to a valid
int x = 3;                expression.
int y = 4;                Note that we did not
int z = 5;                surround the token_list
  SWAP(x,y);              with parenthesis since we
if (x<0)                  do not expect anyone to
   SWAP(x,y);             pass an expression into
else                      SWAP
   SWAP(x,z);
```

Reference: http://cprogramming.com/tutorial/cpreprocessor.html

Backward Macros Subtleties

- So I should ALWAYS use: **do { … } while (0)** macros?
  - No. In general only use do/while for functional macros where you are likely to encounter the if/then/else cases identified earlier.
- Good:
  - `#define SWAP(a,b) do {a^=b; b^=a; a^=b;} while (0)`
- **Bad**:
  - `#define DEF_VAR(i) do { int tmp_ ## i } while(0)` **OR**
  - `#define DEF_VAR(i) { int tmp_ ## i }`
  - **Causes the variable to be defined OUT OF SCOPE in the current function**

3. Macros vs. Functions

Macros

- Code “inlined”, program larger
- Faster execution
- Limited control of side effects
- Typeless

Functions

- Function calls, program smaller
- Extra overhead (slower)
- Good control of side effects
- Typed

Use macros only when appropriate

Good use of Macros in C

- There are ***four (4) common "good" uses of macros*** (and many misuses…)
  1. To ***prevent recursive inclusion*** (toggle switch)
  2. To ***define "constants"***
  3. To ***inline code***
  4. For ***debugging***

- These uses are not exclusive and are often combined (more details can be found in ***Numerical Programmign in C***, chapter 2, pp 29-31)

1. Preventing Recursive Inclusion

- All header files (**.h**) should include a macro which gets defined the first time a header is included
- The macro should be named the macro based on the header name (note the underscores)

Example: Header file name: **timers.h**

Macro name: **_TIMERS_H_**

- To achieve the desired result enclose the entire contents of the header in a conditional compilation directive:

```
#ifndef _TIMERS_H_
#define _TIMERS_H_
 /* contents of header timers.h */
#endif _TIMERS_H_
```

Why Prevent Recursive Inclusion?

- A macro can only be defined once, or a “duplicate definition” error will be generated
  - Even if the definition is identical

- When working on a programming team it is very easy (and encouraged) for a programmer to reuse the work of others
  - Easily resulting in the same low level libraries being included over and over.

- Always prevent macro recursion

```
 #ifndef _TIMERS_H_
 #define _TIMERS_H_
/* contents of header timers.h */
 #endif _TIMERS_H_
```

2. Defining “Constants”

Useful constants from the C standard library **`math.h`**

```
#define MAXFLOAT 3.40282347e+38F
#define M_E      2.7182818284590452354
#define M_LOG2E  1.4426950408889634074
#define M_LOG10E 0.43429448190325182765
#define M_LN2    _M_LN2
#define M_LN10   2.30258509299404568402
#define M_PI     3.14159265358979323846
#define M_TWOPI  (M_PI * 2.0)
#define M_PI_2   1.57079632679489661923
#define M_PI_4   0.78539816339744830962
#define M_3PI_4  2.35619449019234483700E0
#define M_SQRTPI    1.77245385090551602792981
#define M_1_PI   0.31830988618379067154
#define M_2_PI   0.63661977236758134308
#define M_2_SQRTPI       1.12837916709551257390
#define M_SQRT2  1.41421356237309504880
#define M_SQRT1_20.70710678118654752440
#define M_LN2LO     1.90821492927058770002E-10
#define M_LN2HI     6.93147180369123816490E-1
#define M_SQRT3  1.73205080756887719000
#define M_IVLN10    0.43429448190325182765   /* 1 / log(10) */
#define M_LOG2_E    _M_LN2
```

3. Inlining Code

- Example: From **`gsl_complex.h`**

```
#define GSL_SET_COMPLEX(zp,x,y) do{ (zp)->dat[0]=(x); (zp)->dat[1]=(y); } while(0)
#define GSL_SET_REAL(zp,x) do{ (zp)->dat[0]=(x); } while(0)
#define GSL_SET_IMAG(zp,y) do{ (zp)->dat[1]=(y); } while(0)

#define GSL_SET_COMPLEX_PACKED(zp,n,x,y) \
      do {*((zp)+2*(n))=(x); *((zp)+(2*(n)+1))=(y);} while(0)
```

- Why do they enclose the macros in

  `do{ }while(0)`

- ***Ans:*** Allows us to write **"multistatement macros"** that expand to a **"single statement"**

(see example sequence on "macros subtleties" . . .)

4. Comment out Code

- Sometimes we want to “comment out” a chunk of code for debugging.
- To do this the **`#if`** … **`#endif`** macros is handy

```
#if 0
   /* comment out for debugging */
   *ptr = num/maloc(ex*sizeof(nada)
   . . .
#endif
```

VERY Handy!

- Why not just use **`/*  …  */`** ?

The C Preprocessor

mycube

```
#include “wasi”
quispy cube(x):
   tiki manca(x)
Pacha canchis(8)
```

wasi

```
#define quispy def
#define canchis(x) cube(x)
#define manca(x) (x*x*x)
#define Pacha print
#define tiki return
```

Run the C preprocessor

```
cpp –P mycube -o mycube.py
```

Mycube.py

```
def cube(x):
   return (x*x*x)
print cube(8)
```

Compilation Process

Preprocessing stage (-P)

`gcc –E –P gcc_ex01.c` or `cpp –P gcc_ex01.c`

The preprocessor runs and does a physical substitution of macro symbols. Note the A and B

```
/* C compilation with GCC */
#if INC
  #include <stdio.h>  /*for printf*/
#endif
#define A (4)
#define B (5)
int main() {
  printf(“A = %d\n”,A);
  printf(“B = %d\n”,B);
  printf(“A+B = %d\n”,A+B);
  return 0;
}
```

```
/* After gcc –E –P gcc_ex01.c*/
int main()
{
    printf("A = %d\n",(4));
    printf("B = %d\n",(5));
    printf("A+B = %d\n",(4)+(5));
    return 0;
}
```

Compilation Process

Define command line values (-Dxxx)

```
gcc –DINC gcc_ex01.c
```

Define the macro value INC, can be used to control program options.

```
/* C compilation with GCC */
#if INC
  #include <stdio.h>  /*for printf*/
#endif
#define A (4)
#define B (5)
int main() {
  printf("A = %d\n",A);
  printf("B = %d\n",B);
  printf("A+B = %d\n",A+B);
  return 0;
}
```

```
Without –DINC generates:
warning: incompatible implicit
declaration of built-in
function 'printf'

With –DINC: no errors

Hint: #if INC - a macro form of
if/then/else
```

Debugging Macros

- Inspect the output of the preprocessor

- Use the compiler directive –E –P
  - P = expand macros
  - E = stop after expansion

```
gcc –E –P mycode.c > foo
```

Macros Summary

- ***Generally we don't put semicolon at the end of a macro.***
- Surround macro arguments and macro definitions with parenthesis.
- Adopt naming convention (capitals, underscores, etc.) that makes it "obvious" when a macro is used.
- Avoid using macros to "create" a new language.
- Macros are used extensively in C programming.

Problem 1

- Which macro is the “best”, why?
  1. #define SQ(x) x * x
  2. #define SQ(x) (x) * (x)
  3. #define SQ(x) ((x) * (x))
  4. #define SQ (x) ((x) * (x))
  5. #define SQ(x)((x) * (x));

- #3 is the best, it doesn’t have an extra space, it doesn’t have any ending semicolon and it has extra parenthesis

Problem 2

- Your C code includes the following, what will the compiler do?

```
#include <stdio.h>
#include “stdlib.h”
#include <myheader.h>
```

- The compiler will search for:
  - Stdio.h in the system library and find it.
  - Stdlib.h in the current directory and then the system library and will find it.
  - myheader.h in the system library and WILL NOT FIND IT

Problem 3

- What do the following macro commands do and when would you use them?

  _ _FILE_ _

  _ _LINE_ _

`_ _FILE_ _` Source file name being compiled

`_ _LINE_ _` Current line number in file

- These are often used in error and debug messages because they are automatically updated as the C file is changed

Problem 4

- Which macro is better and why?
  1. **#define DEC_VAR(V) int _t_ ## V;**
  2. **#define DEC_VAR(V) int V;**

- #1 is the best.
  - By adding the _t_ to the front of the variable, the programmer will know that that variable is a macro variable and not a “real” variable in the debugger.
  - The _t_ also help protect the programmer from multiple defined variable errors because the main code is not going to have an _t_ in it.

Problem 5

- Is this a legal macro?
- What comments do you have about it?

```
#define FOR_START(R)   for(i = 0; i < (R); i++)
```

- The macro is legal.
  - A macro can be any part of the C language.
  - This kind of macro is considered poor form but it is sometimes required.

Problem 6

- What will happen?

```
#include “myheader.c”
```

- The compiler will **“copy in” myheader.c**
  - Which is ALL CODE
    - Include files should NEVER include CODE
    - This is not scaleable
    - The C Preprocessor doesn’t know C, it just does what it is told
  - Compile myheader.c separately and LINK IT IN