C preprocessor

Henrik Friedrichsen

Arbeitsbereich Wissenschaftliches Rechnen
Fachbereich Informatik
Fakultät für Mathematik, Informatik und Naturwissenschaften
Universität Hamburg

2013-11-14

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

informatik die zukunft

Outline

1 Introduction

2 Syntax

3 Caveats

4 Performance

5 Summary

6 References

Introduction

Typical order of compilation processes

- Preprocessor statements are interpreted and expanded **before** compilation
- Is not limited to C, can virtually used be for every text-related task
- Preprocessor statements are not *actual* C code but instead instruct the preprocessor to do simple text substitutions
- Statements are rather simple
- Included in the C standard
- Several macros (constants) are already predefined, e.g.:
  - `__DATE__`, `__FILE__`, `__LINE__`, `__TIME__`, ..
  - Platform specific macros: `__LINUX__`, `_WIN32`, ..

Outline

1 Introduction

2 Syntax

3 Caveats

4 Performance

5 Summary

6 References

Syntax (overview)

Syntax and use-cases:

- File inclusion: `#include`
- Conditional compilation: `#if`, `#ifdef`, ...
- Compiler instructions: `#pragma`, `#error`, `#warning`
- Macro definition: `#define`, `#undefine`
- Stringification, Concatenation: `#`, `##` Operator

File inclusion: `#include <file>`

Includes a files content at the position of the statement.
From include path:

```
#include <file.h>
```

or a *local* file:

```
#include "file.h"
```

Conditional compilation: `#if`, `#ifdef`, ...

There are predefined macro definitions. For instance:

- the platform/architecture, compiler (+version)
- mathematical constants (PI in `math.h`)

```
#ifdef __LINUX__
  #include <sys/socket.h>
#elif _WIN32
  #include <winsock.h>
#else
  // other platforms
#endif
```

Compiler instructions: #pragma, #error, #warning

- Used to influence compiler behaviour
- Pass parameters to the compiler per-source-file similar to compilation flags
- Often compiler specific, e.g. not defined by the C standard

Add -O2 (level 2 optimization) to the compiler flags

```
#pragma GCC optimize ("O2") // Compile with -O2
```

.. or initiate an OpenMP environment:

```
#pragma omp ...
```

Generator a compilation error on Windows platforms:

```
#ifdef _WIN32
   #error "Will not compile on Win32. Aborting."
#endif
```

Macro definition: #define, #undefine

Used to define simple *"functions"* or constants:
Object-like macro (often used for constants):

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

Function macro:

```
#define <identifier>(parameters) <replacement>
```

Deleting a macro:

```
#undef <identifier>
```

Macro definition: #define, #undefine

debug.h

```
#ifdef _DEBUG
  #define DPRINT(x)\
    printf("[%s:%i] %s\n", __FILE__, __LINE__, (x))
#else
  #define DPRINT(x)
#endif
```

debug.c

```
#include <stdio.h>
#include "debug.h"
#define EXITCODE_OKAY 1 // macro as a constant

int main(int argc, char* argv[])
{
  DPRINT("verbose debugging output");
  return EXITCODE_OKAY;
}
```

Macro definition: #define, #undefine

Testing out example (with and without _DEBUG defined):

```
% gcc debug.c
% ./a.out
% gcc debug.c -D_DEBUG # macros can also be defined with cmdline
    parameters
% ./a.out
[debug.c:7] verbose debugging output
```

Macro definition: `#define`, `#undefine`

`assert(exp)` defined in **`assert.h`** works similar to this:

assert(3) from the Linux Programmer's Manual (manpage)

If the macro **`NDEBUG`** was defined at the moment **`<assert.h>`** was last included, the macro **`assert()`** generates no code, and hence does nothing at all.
Otherwise, the macro **`assert()`** prints an error message to standard error and terminates the program by calling **`abort(3)`** if expression is false (i.e., compares equal to zero).

Expressions that change the environment can lead to ***heisenbugs***.

Example: **`assert(FreeResources())`**

Stringification, Concatenation: #, ## Operator

Stringification: (*converts a parameter to a string literal*)

```
#define LOG_COND(cond)\
  printf("expression "#cond" is: %i\n", (cond))

LOG_COND((2 + 4 == 6));
```

Results in: **`expression (2 + 4 == 6) is:  1`**

Concatenation: (*concatenate macro tokens*)

```
#define HEX(v) (0x##v)

printf("value: 0x%X\n", HEX(DEF));
```

Results in: **`value:  0xDEF`**

Outline

1. Introduction
2. Syntax
3. Caveats
4. Performance
5. Summary
6. References

Caveats

Things to watch out for:

- Double evaluation
- Debugging
- Forseeability
- Operator Precedence

Double evaluation

Due to the nature of preprocessor macros the programmer can run into difficult situations. **For instance**:

```
#include <stdio.h>

#define MAX(a, b) (a > b ? a : b)

int main(int argc, char* argv[]) {
  int apples = 11, kiwis = 12;

  printf("Maximum value MAX(kiwis=%i, apples=%i): %i\n",
      kiwis, apples, MAX(kiwis, apples));

  printf("Add one more apple MAX(kiwis, ++apples): %i\n",
      MAX(kiwis, ++apples));

  printf("Err.. what? Kiwis: %i, Apples: %i\n",
      kiwis, apples);

  return 1;
}
```

Double evaluation

Result:

```
% ./double_eval
Maximum value MAX(kiwis=12, apples=11): 12
Add one more apple MAX(kiwis, ++apples): 13
Err.. what? Kiwis: 12, Apples: 13
```

The expression passed as **b** is evaluated and executed twice:

```
#define MAX(a, b) (a > b ? a : b)

-> MAX(kiwis, ++apples) in MAX(a > b ? a : b)
--> (kiwis > ++apples ? kiwis : ++apples)
```

Passing function calls is problematic as well! Why?

Double evaluation (workaround)

A workaround would be to instantiate variables for the parameters like such:

```
#define max(a,b) \
  ({ typeof (a) _a = (a); \
     typeof (b) _b = (b); \
     _a > _b ? _a : _b; })
```

Example taken from the GNU GCC extensions manual

`typeof` is a GCC extension and **not** part of the C standard.

Debugging

Debugging can be painful when the code uses a lot of macros.
Why?

- Macros are expanded, meaning the code changes before/during compilation.
- Some debuggers are unable to process this
- Locating bugs can prove difficult

Forseeability

It is hard for the developer to *oversee* the effects of macro expansion. This can lead to problems:

```
#include <stdio.h>

#define LOG_NULLPTR(x)\
  if(x == NULL) printf("is a nullptr!\n")

int main(int argc, char* argv) {
  void *foo = (void*)1;

  if(1)
    LOG_NULLPTR(foo);
  else
    printf("condition is not true!\n");
}
```

Which will *actually* result in:

```
void *foo = (void*)1;
if(1)
  if(foo == NULL)
    printf("is a nullptr");
  else
    printf("condition not true!\n")
```

Forseeability

To prevent situations like this make sure that the code generated by the macro will not influence the code where it is included.

One technique is to wrap the code in it’s own block.
For example by putting it in a loop that will only iterate once:

```
#define LOG_NULLPTR(x)\
  do {\
    if(x == NULL) printf("is a nullptr!\n"); \
  } while(0)
```

Operator Precedence

Caution is required when using mathematical expressions in macros.

Assume we have this definition of **CUBE**(x):

```
#define CUBE(x) x*x*x // yes, it's vulnerable to double eval
```

The following examples demonstrate a few problems:

```
CUBE(2 + 2)  // Expected: (2+2)^3 = 64
=> 2 + 2*2 + 2*2 + 2 = 12

5*CUBE(4 - 3) // Expected: 5*((4-3)^3) = 5
=> 5*4 - 3 * 4 - 3 * 4 - 3 = -7
```

Fix: Use parentheses around parameters as well as whole macro result

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

CUBE(2 + 2)  // Expected: (2+2)^3 = 64
=> ((2+2)*(2+2)*(2+2)) = 64

5*CUBE(4 - 3) // Expected: 5*((4-3)^3) = 5
=> 5*((4-3)*(4-3)*(4-3)) = 5
```

Outline

1 Introduction

2 Syntax

3 Caveats

4 Performance

5 Summary

6 References

Improving performance

Can we improve performance with the aid of C preprocessor macros?

If so, how?

- `inline` keyword vs. usage of macros
  - `inline` keyword is only a **suggestion** to the compiler
  - With the help of macros one can **force** the compiler to inline code, because in reality there is no function call.

Reducing Stack Overhead

Example: Forcing code-inlining with macros

```
#include <stdio.h>
#include <limits.h>
#include <math.h>

#ifndef _INLINE
  double veclength (double x, double y, double z)
  {
    return sqrt(x*x + y*y + z*z);
  }
#else
  #define veclength(x, y, z) sqrt((x)*(x) + (y)*(y) + (z)*(z))
#endif

int main(int argc, char* argv[]) {
  int i;
  for(i = 0; i < INT_MAX; i++)
    veclength(150.5, 200.5, 300.0);
  return 1;
}
```

Performance differences

Significant difference in this case:

```
% gcc perf.c -lm
% time ./a.out
./a.out  23.50s user 0.00s system 99% cpu 23.513 total
% gcc perf.c -lm -D_INLINE
% time ./a.out
./a.out  7.92s user 0.00s system 99% cpu 7.931 total
```

23.5 seconds vs. **8** seconds!

- However: A very constructed case
- Only significant when we have a lot of stack overhead due to function calls
- Why? Reduction of severe overhead, for instance:
  - Pushing arguments to the stack
  - Grabbing arguments off the stack
  - Pushing/popping return-address (at least for x86 **`call`**)
  - Location jumps

Tradeoff

However, it’s *not always* this simple. A lot of other factors play a role:

- With inlining the code size increases
  - Code size might be larger than instruction cache
  - → Instruction cache miss
  - Code needs to be loaded into construction every time
  - → Performance loss?
- This usually applies to large/complex functions
- In most cases the compiler is smart (*or smarter*) enough to determine whether functions should be inlined

Results in a **Tradeoff** between *code size* and *function call* overhead.

Summary

- Text substition like functionality
- Can serve as a handy tool to simplify code
- Access to platform/compile(-time) information (useful for portability)
- To be used with care (see *caveats*)
- Performance improvement of code is possible by *force-inlining* code

References

- **Mainly**: GNU GCC Documentation
- Linux @ CERN
- GNU/Linux manpages