Usages of Macro of C Programming Language such as Prefix Style Operator

Hirofumi NAKAMURA[1] and Takayasu FUCHIDA[2]

(Accepted October 1, 2019)

Abstract Language processors of C programming language and C++ programming language have preprocess functions for the compilation. One of the preprocess functions, called "macro," can do replacement alphanumeric names and their arguments to defined strings and the arguments in the source code. This paper shows that the instructions of execution controls of statements and expressions can be expressed such as prefix style operators using macros in certain situations of source code writen in C or C++ programming language. Specifically, (1): iteration controls and (2): execution/non-execution controls are described. (1) leads to describing concise and easy-to-understand code by appropriately naming macros. (2) leads to reductions in keyboard operations related to trial and error. Furthermore, unlike commenting, since the compilation check is also done on non-executing parts on (2), there is a possibility to notice early that the non-executable part needs to be modified.

Keywords [C programming language, C++ programming language, Preprocessor, Macro, Prefix style operator]

1 Introduction

Language processors of C programming language[1)] (hereinafter briefly called C and including C++ programming language[2)], hereinafter briefly called C++, which has upper compatibility with C) have preprocess functions for the compilation. One of the preprocess functions, called "macro," can do replacement (called macro expansion) alphanumeric strings (called macro names) and their arguments to defined strings and the arguments in the source code (hereinafter called "code"). Each defined replacement which has alphanumeric name and replacement rule is also usually called macro.

Using macros bring (a): partial omission in code description[3, 4)], (b): syntax abstraction[5)] and (c): omission of overhead of function call[3, 6)]. In this paper, we propose several examples of effective macro uses contribute for (a) and (b).

Specifically, in section 2, we show examples so that macros with meaningful names can briefly describe instruction of repetitive control in a code such as prefix style operator (hereinafter briefly called "operator").

Furthermore, in section 3, we show examples so that macros can briefly describe the instructions of execution/non-execution control in the code on the trial and error or debugging during a program development.

We compare our method with existing similar uses of macros in Ref. 7) and the Linux code[8)] etc. in both sections.

2 Instruction of iteration such as operator

2.1 Concrete examples

There are cases where the instructions of iterations can be written such as operators.

1 General Education Division, National Institute of Technology(KOSEN), Miyakonojo College
2 Graduate School of Science and Engineering, National University Corporation Kagoshima University

At first, we show application examples to the following code[†1] by the sweeping-out method for a inverse matrix.

```
for(i=0; i<N; i++)
  for(j=0; j<N; j++)
    inv[i][j] = ((i==j) ? 1.0 : 0.0);

for(i=0; i<N; i++){
  w = 1.0/a[i][i];
  for(j=i+1; j<N; j++) a[i][j] *= w;
  for(j=0; j<N; j++) inv[i][j] *= w;
  for(j=0; j<N; j++){
    if(i!=j){
      w = a[j][i];
      for(k=i+1; k<N; k++)
        a[j][k] -= a[i][k]*w;
      for(k=0; k<N; k++)
        inv[j][k] -= inv[i][k]*w;
    }/* if */
  }/* j */
}/* i */
```

For this code, if you define[†2] macros such as

```
#define  F(v)  for(v=0; v<N; v++)
#define  Fi  F(i)
#define  Fj  F(j)
#define  Fk  F(k)
```

with `F` in common or

```
#define  Fi   for(i=0; i<N; i++)
#define  Fj   for(j=0; j<N; j++)
#define  Fk   for(k=0; k<N; k++)
```

with no other common macro, the code can be described[†3]as follows[†4] .

```
Fi Fj inv[i][j] = ((i==j) ? 1.0 : 0.0);

Fi{
  w = 1.0/a[i][i];
  for(j=i+1; j<N; j++) a[i][j] *= w;
  Fj inv[i][j] *= w;
  Fj{
    if(i!=j){
      w = a[j][i];
      for(k=i+1; k<N; k++)
        a[j][k] -= a[i][k]*w;
      Fk inv[j][k] -= inv[i][k]*w;
    }/* if */
  }/* Fj */
}/* Fi */
```

Whether or not to define macros used fewer times is arbitrary. However, if you define further macro such as

```
#define  Fij  Fi Fj
```

with `Fi` and `Fj` , the beginning of the code is also possible to describe as follows, where naming of macros and how to put spaces and line breaks in the code are arbitrary.

```
Fij inv[i][j] = ((i==j) ? 1.0 : 0.0);
```

In the above examples, a character `F` reminiscent of "For each" is used in `F` and `Fi` etc.

Instead of `F` portion of the macro name `Fi` etc., using `A` reminiscent of "All" or `E` reminiscent of "Each" is arbitrary, for exmaple `A`, `Ai`, `Aj` and `Aij`.

Compared to using only `F(…)`, using `Fi` and `Fj` etc. is more concise. However, using `F(…)` is also arbitrary.

When iterations whose final values of individual control valuables are different are used, for example, if M

†1 For example, we assume `#define N 100` , `double a[N][N], inv[N][N], w;` and `int i,j,k;` .
†2 Where, to define control variables or not in iteration statements is arbitrary.
†3 For your information, if you define a macro that can describe array `a[i][i]` such as `a(i,j)` or `aij` (fot examples, by with `#define a(i,j) a[i][j]` or `#define aij a[i][j]`), the code will be more concise and closer to the mathematical expression. Further more, when you want to check the ranges of array index's values on debugging, you can finish by only one place modification in the macro definition such as `#define a(i,j) a[ (0<=(i)&&(i)<MAX)?(i):printf("ax:%d\n",(i)) ][…]` instead of writing same strings throughout the code.
†4 Specific examples of the proposed methods are written in the blue square frames.

times repetition is also used, some macros defined such as

```
#define  FvV(v,V)  for(v=0; v<V; v++)
#define  FiN  FvV(i,N)
#define  FjN  FvV(j,N)
#define  FiM  FvV(i,M)
#define  FjM  FvV(j,M)
```

or

```
#define  AvV(v,V)  for(v=0; v<V; v++)
#define  AiN  AvV(i,N)
#define  AjN  AvV(j,N)
#define  AiM  AvV(i,M)
#define  AjM  AvV(j,M)
```

can be used. The macro names of here, such as `AiN`, are examples of namings reminiscent of universal quantifier such as $\forall i \in \{k \,|\, 0 \leq k < N,\ k \in \mathbb{N}\}$.

The following example is the tracing of a linear list. If you define macros such as[†5]

```
#define  Trace(data, v)  \
  for(v=data##HEAD; v!=NULL;  \
      v=v->next)
#define  Tp  Trace(list, p)
```

or a macro such as

```
#define  Tp  \
  for(p=listHEAD; p!=NULL; p=p->next)
```

by combining, the trace of the list can be described with `Tp` as follows.

```
sum=0;  Tp sum += p->value;
```

When multiple linear lists (in general, similar plural data structures) are used, it is arbitrary to selectively use macros named with strings of the parts of each list's name. In particular, for the data whose repeat structures are same, it is arbitrary using a common macro so that each data name is specified as an argument of the common macro as follows.

```
#define  Trace(data, v)  \
  for(v=data##HEAD; v!=NULL;  \
      v=data[v].next)
#define  Tp(diff)  Trace(list##diff, p)
```

It can be used as follows.

```
sum1=0;  Tp(1) sum1 += list1[p].value;
sum2=0;  Tp(2) sum2 += list2[p].value;
```

As described above, you can write instructions of iterations such as prefix style operators when you usually write iterations with "for" or "while" statements. When the same or similar repetitive control appears many times, the advantage of simplifying the description increases. In addition, there is also an advantage of macros that it may be possible to make a correction at only one place, not at each place, when a common correction is required in the code.

There is a function definition besides a macro definition as C's function that makes it possible to write the code parts, that appears several times in code, in the form of a name and arguments. Since the definitions of named function cannot be described inside a function definition, even when the use points of the functions are limited to a relatively narrow range of the entire code, the positions of the function definition may be far away from the use points. This must be a big burden when viewing the code. Because macros can be defined also in the middle of a function definition, you can choose the positions of the function definitions near where you use them.

†5 Here, backslash at the end of the line means continuation of lines. The same applies to the following.

2.2 Existing similar examples

Macros of code part[†6]

```
#define loop(n) {int _i_;  \
  for(_i_=1;_i_<=(n);_i_ ++) {
#define lend }}
```

in the Ref. 7) used by sandwiching the object to be repeated between the front and the back. On the other hand, the proposed description method given in 2.1 does not require posterior parts. Furthermore, since

```
#define forto(i,from,to)  \
  for(i=(from);i<=(to);i++)
#define downto(i,from,to)  \
  for(i=(from);i>=(to);i--)
```

in Ref. 7) stay in general-purpose with almost all information instructed as arguments, the proposed description method given in 2.1 is simpler.

As examples from a large-scale C code involving a lot of people, we looked for iteration macro definitions using “for” or “while” statements in Linux code[8)]. For examples, there exists definitions such as

```
#define for_each_ethrxq(sge, i)  \
  for (i = 0; i < (sge)->ethqsets; i++)
```

in the file drivers\net\ethernet\chelsio\cxgb4\cxgb4.h,

```
#define for_each_pci_dev(d)  \
  while ((d = pci_get_device(PCI_ANY_ID, \
    PCI_ANY_ID, d)) != NULL)
```

in the file include\linux\pci.h,

```
#define shdma_for_each_chan(c, d, i)  \
  for (i = 0, c = (d)->schan[0];  \
       i < (d)->dma_dev.chancnt;  \
       c = (d)->schan[++i])
```

in the file include\linux\shdma-base.h,

```
#define gmap_for_each_rmap(pos, head)  \
  for (pos = (head); pos; pos = pos->next)
```

in the file arch\s390\include\asm\gmap.h and

```
#define LOAD_FIXED_STATE(tbl,dev)  \
  for (i = 0;  \
    i < sizeof(tbl##Table##dev)/8; i++)  \
  chip->dev[tbl##Table##dev[i][0]]  \
    = tbl##Table##dev[i][1]
```

in the file drivers\video\fbdev\riva\riva_hw.c. Their names include information in not short names to make them easier to understand in the big code. Compared to them, we emphasize that proposed description method such as in 2.1 is simple prefix style operator.

3 Instruction of execution/non-execution of code part such as operator

In trial and error or debugging during the program development, some code parts may become necessary or may become unnecessary. It may be common to do them by commenting and uncommenting.

About this, examples of instructions such as operators are given in following 3.1 and 3.2.

3.1 Instruction of execution/non-execution for one statement

At first, we treat descriptions such as operators for control of execution/non-execution of one statement.

It is often done to make comment such as

```
proc_a(1); /*proc_b(2);*/ proc_c(3);
```

with `/*` and `*/`. On the other hand, if you prepare a macro definition such as

```
#define  D  if(0)
```

by using “if” statement, you can write as follows[†7].

†6 The following examples are input without pares of a backslash and a line break in Ref. 7) and Ref. 8).

†7 There exists definitions of the same purpose using ternary operator in http://tricky-code.net/nicecode/code10.php (referred at 2018-7-11) of tricky-code.net such as `#define debug` and `#define debug 1 ? (void)0 :`. However, ternary operator can be applied to expressions in general, but it cannot be used for statements such as “for”, “while” and compound statements enclosed in { and }, causing errors.

```
proc_a(1); D proc_b(2); proc_c(3);
```

In this example, although the name of the macro is assumed to be temporarily D, naming is actually arbitrary. During a program development, when the reversal of execution and non-execution control is frequent, the change with our method will be efficient because characters to change are few. Although macro name can be more than two characters, one character is optimal.

Compared with ordinary non-execution enclosed by /* and */, macro D can non-execute one statement only by inputting two characters such as a D and a space. Furthermore, it is not necessary to move the input cursor to the end point of the code portion must be non-executed. If a compound statement enclosed by { and } is targeted, you don't need a space before {, so you only need to enter one character D.

The effort of the undoing is also efficient, i.e. remove a D and a space or remove only a D, than uncommenting which removes /* and */.

In the C ++ code, // is convenient for commenting up to the end of the line. However, if you want to switch between executing and non-executing for one statement in the middle of a line, not until the end of the line, the macro such as above D is more convenient also in the C++ code.

3.2 Instruction of execution/non-execution for multiple statements or multiple lines

If you want to apply a macro such as D to multiple statements in one line or in multiple lines, you can do it such as

```
D{
 :
 :
}
```

with D, { and }.

Inversions of executions and non-executions of the code parts are possible by inputting or deleting the character D. However, if you prepare by adding a macro

```
#define  DD  if(1)
```

or a macro

```
#define  DD
```

whose macro expansion is an empty string, insert strings in each place of the code are always one character, i.e. D. If the inversions are frequent, this method is more beneficial. It is arbitrary whether to prepare furthermore macro such as

```
#define  DDD  if(0)
```

with one more character.

To delete Ds in the code may be one way for activating. However, the use of DDs has the advantage that the signs that it is in the process of improvement of the code do not disappear. The macro names are arbitrary, such as D and E, X and EX[†8], D and ND, and modified macro names with the purpose or group etc., instead of D and DD.

When you want to activate all Ds, of course, you have the choice to modify the definition of D to

```
#define  D  if(1)
```

instead of deleting all D in the code. Otherwise, you have the choice to modify the definition of D to

```
#define  D
```

which is expanded to an empty string. These complete only one portion change in entire the code.

In the above, D and DD work such as an operator acting on the next lump.

†8 X is chosen by an association from no good of Japanese sense, EX is chosen by an association from execute.

Since D and DD are actually realized by "if" statements, the effects of D and DD can be interrupted by inserting ; immediately after them. That is, as another way of activation, the instruction can be done by inputting ; and making it as D;, instead of deleting D. The notes on the way are described in 3.4.

On the controls of execution/non-execution of single or multiple lines of a code, it is often surrounded by #if line and #endif line. For example, debug related code parts are surrounded such as follows.

```
#define DEBUG_SW_xxxxx  0

#if DEBUG_SW_xxxxx
  :
  :
#endif
```

On the other hand, it is conceivable to prepare a macro such as

```
#define  DEBUG_xxxxx  if(0)
```

or

```
#define  DEBUG_xxxxx  if(1)
```

and to rewrite with { and } as follows.

```
DEBUG_xxxxx {
  :
  :
}
```

In this example, DEBUG_xxxxx is used as the macro name. DEBUG_xxxxx can be regarded as an operator that determines execution/non-execution of the debug related code parts.

For non-execution of unused code parts or checking code parts, commenting or surrounding with #if 0 and #endif are often done. However, these may cause the following disadvantages. If there are changes in the data structures etc., some reflections may be forgotten in the commented parts. And when you want to execute them suddenly at a later date, after the compiling errors occured after the uncommenting or replacement of #if 0 to #if 1, you can notice for the first time that they have not been reflected. To make corrections after a long time will increase the time to remember and confirm.

However, if they had not been commented, they could be noticed and started correspondence with other related parts at the time of the first compilation with errors after changing the data structures etc. if not sooner. Even if execution is not done, but the compilation check is done, you will increase the possibility that you will notice the inconsistency as soon as possible. The use of macros is also beneficial in this point.

3.3 Existing similar examples

There exists similar purpose code parts in Linux code[8] such as

```
#ifdef CONFIG_NET_CLS_ACT
#define tcf_exts_for_each_action(i, a,  \
                                 exts)  \
  for (i = 0;  \
       i < TCA_ACT_MAX_PRIO && ((a)  \
       = (exts)->actions[i]); i++)
#else
#define tcf_exts_for_each_action(i, a,  \
                     exts)  \
  for (; 0; (void)(i), (void)(a),  \
                       (void)(exts))
#endif
```

in the file include\net\pkt_cls.h,

```
/* #define apic_debug(fmt,arg...)
  printk(KERN_WARNING fmt,##arg) */
#define apic_debug(fmt, arg...)  \
  do {} while (0)
```

in the file arch\x86\kvm\lapic.c,

```
#ifdef MMU_DEBUG
:
#define pgprintk(x...)  \
  do { if (dbg) printk(x); } while (0)
```

```
#define rmap_printk(x...)  \
  do { if (dbg) printk(x); } while (0)
#define MMU_WARN_ON(x) WARN_ON(x)
#else
#define pgprintk(x...) do { } while (0)
#define rmap_printk(x...) do { } while (0)
#define MMU_WARN_ON(x) do { } while (0)
#endif
```

in the file arch\x86\kvm\mmu.c,

```
#ifndef CONFIG_LDM_DEBUG
#define ldm_debug(...) do {} while (0)
#else
#define ldm_debug(f, a...)  \
  _ldm_printk (KERN_DEBUG, __func__,  \
                              f, ##a)
#endif
```

in the file block\partitions\ldm.c,

```
#if defined(QL_DEBUG)
#define DEBUG(x)   do {x;} while (0);
#else
#define DEBUG(x) do {} while (0);
#endif
```

in the file drivers\scsi\qla4xxx\ql4_dbg.h,

```
#ifdef DEBUG
#define pr_debug(format, ...)  \
  fprintf(stderr, format, ## __VA_ARGS__)
#else
#define pr_debug(format, ...)  \
  do {} while (0)
#endif
```

in the file tools\virtio\virtio-trace\trace-agent.h,

```
#ifdef CONFIG_MEMCG_KMEM
:
#define for_each_memcg_cache_index(_idx) \
  for ((_idx) = 0; \
    (_idx) < memcg_nr_cache_ids;  \
    (_idx)++)
:
#else
:
#define for_each_memcg_cache_index(_idx) \
  for (; NULL; )
:
#endif /* CONFIG_MEMCG_KMEM */
```

in the file include\linux\memcontrol.h,

```
#ifdef DEBUG
#define  DBG(x...) printk(x)
#else
#define  DBG(x...) do { } while (0)
#endif
```

in the file arch\arm\plat-iop\pci.c,

```
#ifdef DEBUG
#define ASSERT(x)  \
do {  \
  if (!(x)) {  \
    printk(KERN_EMERG "assertion failed  \
       %s: %d: %s\n",  \
      __FILE__, __LINE__, #x);  \
      BUG();  \
  }  \
} while (0)
#else
#define ASSERT(x) do { } while (0)
#endif
```

in the file arch\x86\kvm\ioapic.h. These macros are not prefix style and the arguments of them will not be checked at the compilation on a selection of their definitions.

The macros defined at the code part

```
#ifdef DEBUG
#define dprintf printf
#else
#define dprintf(...) do { } while (0)
#endif
```

in the file tools\power\cpupower\bench\config.h can be used only for output.

The macros definition at the code part

```
#ifdef NDEBUG
#define BUG_ON(cond)  \
  do { if (cond) {} } while (0)
```

```
#else
#define BUG_ON(cond) assert(!(cond))
#endif
```

in the file tools\include\linux\kernel.h and

```
#ifdef VERBOSE_DEBUG
#define COH_DBG(x) ({ if (1) x; 0; })
#else
#define COH_DBG(x) ({ if (0) x; 0; })
#endif
```

in the file drivers\dma\coh901318.c are not prefix style.

The macros definition at the code part

```
static inline void activate_mm(  \
  struct mm_struct *active_mm,
  struct mm_struct *mm)
{
  get_mmu_context(mm);
  set_context(mm->context, mm->pgd);
}
```

in the file arch\m68k\include\asm\mmu_context.h and

```
#define deactivate_mm(tsk, mm)  \
  do { } while (0)
#define activate_mm(prev, next)  \
  switch_mm(prev, next, NULL)
```

in the file arch\unicore32\include\asm\mmu_context.h need keyboard operations at all place the macro used when switching because they have different names for execution and non-execution.

3.4 Points to be noted

When macros such as `D`, `DD` and `DEBUG_xxxxx` are used, they may change the scope of control of "if" syntax and "else" syntax. Even if the macro name does not include the string `if`, do not forget that "if" statement is used for realization.

And, if macros such as `D`, `DD` and `DEBUG_xxxxx` are under control of the preceding control syntax, for example "for", "while", "if" and "else" statements, you should not block their effects by inserting `;`.

When macros such as `D` and `DD` for execution/non-execution control remain in the code after trial and error or debugging, it should be noted that the code is not usually said to be a "beautiful code."

In scenes where you want to make executable programs smaller, such as in product versions, keep in mind whether the suppression of compiler's dead code generation for `if(0)` is default or explicit setting.

4 Conclusions

We proposed that (1): iteration instructions and (2): code part's execution/non-execution instructions can be described as prefix style operators using the macro of C programming language and C++ programming language.

(1) leads to describing concise and easy-to-understand code by appropriately naming macros. (2) leads to reductions in keyboard operations related to trial and error. Furthermore, unlike commenting, since the compilation check is also done on non-executing parts on (2), there is a possibility to notice early that the non-executable part needs to be modified.

Although these are effective methods, applicable scenes do not always exist, and they are hard to be used when they do not match the choosy, the customs or the styles of individual program creators.

This paper is based on the conference presentation at the 81st National Convention of Information Processing Society of Japan[9].

References

1) Kernighan, B.W. and Ritchie, D.M., Translated by Ishida, H.: プログラミング言語 C (original title is "The C Programming Language"), Kyoritsu Shuppan(1989), (in Japanese).
2) Stroustrup, B., Translated by Nagao, T.: プログラミング言語 C++(original title is "The C++ Programming Language"), ASCII(1998), (in Japanese).
3) Tokawa, H.: ザ・C(meaning "the C") 2nd edition 9th impression, SAIENSU-SHA(2004), (in Japanese).
4) Mukuda, M.: はじめての C (meaning "C for the first time") 4th edition, 7th impression, Gijutsu-Hyohron(2005), (in Japanese).
5) Kernighan, B., Bentley, J. and Matsumoto, H., Translated by Kuno, Yo. and Kuno, Ya.: ビューティフルコード (original title is "Beautiful Code") 1st edition, 3rd impression, O'Reilly Japan(2008), (in Japanese).
6) Noro, H., Matsumoto, N.: Linear Algebra by C

Language(Japanese title is “C 言語による線形計算”) (1), Implementation of MATRIX Data Type and Fundamental Matrix Manipulation Functions(Japanese subtitle is “行列のためのデータ型の実現と基本的な行列操作関数”), Geoinformatics, Vol.3, No.4, pp.211–217(1992), (in Japanese).
7) Hayashi, H.: C プリプロセッサ・パワー (meaning “C preprocessor power”) 4th impression, SoftBank(1989), (in Japanese).
8) The Linux Kernel Organization: linux-5.2.11, The Linux Kernel Archives, https://www.kernel.org/ (referenced at 2019-8-29).
9) 中村博文, 渕田孝康: C言語マクロを前置型の作用素のように使うことについて (meaning “Usage of the Programming Language C’s Macro such as Prefix Operator”), 情報処理学会第 81 回全国大会 (The 81st National Convention of Information Processing Society of Japan), 2B-06, pp.1-161 – 1-162 (March 2019), (in Japanese).