# รูปแบบและมาตรฐานการเขียนโปรแกรมภาษาซี

Recommended C Style and Coding Standards

**ดร. พีระพนธ์ โสพัศสถิตย์**

**ศูนย์เทคโนโลยีอีเล็กทรอนิกส์และคอมพิวเตอร์แห่งชาติ**

# Acknowledgments

Much appreciation are due to the authors of *Recommended C Style and Coding Standards*, namely, L.W. Cannon, R.A. Elliott, L.W. Kirchhoff, J.H. Miller, J.M. Milner, R.W. Mitze, E.P. Schan, and N.O. Whittington of Bell Labs; Henry Spencer of Zoology Computer Systems, University of Toronto; David Keppel, EECS, UC Berkeley/CS&E, University of Washington; Mark Brader of SoftQuad Incorporated, Toronto. The stupendous repertoire and years of combined experience of the authors yielded this invaluable guideline for better practice of C programming as described in the abstract of the Style Guide:

> "This document is an updated version of the *Indian Hill C Style and Coding Standards* paper, with modifications by the last three authors. It describes a recommended coding standard for C programs. The scope is coding style, not functional organization."

My special thank goes to David Keppel who grants permission to translate the Style Guide into Thai and whose advice (in our email correspondence) certainly is one novice C programmers should heed:

> **"I think the most important lesson is: code with one consistent style. Even if it's a 'bad' style, at least someone reading the code can pick up the style and then not get further confused by it."**

This Style Guide will not be complete without the famous "*The Ten Commandments for C Programmers*" by Henry Spencer. With permission from Henry Spencer, the original version of The Ten Commandments was replaced by his "*Annotated*" version to end the translated style guide on a high note.

I sincerely hope that this translation will instill many more Thai programmers to become aware of the essence of good programming practices as I myself have long been inspired by numerous research works of the aforementioned authors.

Peraphon Sophatsathit

Bangkok, Thailand

May 1, 1997

# สารบัญ

หน้า


Acknowledgments ..... 2

คำนำ ..... 4

บทนำ ..... 5

1. Introduction ..... 6

2. File Organization ..... 6

3. Comments ..... 9

4. การประกาศตัวแปร (Declarations) ..... 10

5. Function Declarations ..... 12

6. Whitespace ..... 13

7. Examples ..... 14

8. Simple Statements ..... 15

9. Compound Statements ..... 17

10. Operators ..... 20

11. กฎเกณฑ์การตั้งชื่อ (Naming Conventions) ..... 21

12. ค่าคงที่ (Constants) ..... 21

13. Macros ..... 22

14. Conditional Compilation ..... 23

15. Debugging ..... 24

16. Portability ..... 25

17. ANSI C ..... 32

18. ข้อควรคำนึงพิเศษ (Special Considerations) ..... 35

19. Lint ..... 36

20. Make ..... 36

21. มาตรฐานเฉพาะงาน (Project-Dependent Standards) ..... 37

22. บทสรุป (Conclusion) ..... 37

บรรณานุกรม (References) ..... 39

The Ten Commandments for C Programmers ..... 40

# คำนำ

หนังสือที่ท่านถืออยู่ในมือขณะนี้ เรียบเรียงจากต้นฉบับภาษาอังกฤษ ดังปรากฏในกิตติกรรมประกาศ (Acknowledgments) ซึ่งเป็นผลงานโดย “ผู้ทรงคุณวุฒิ” ของวงการภาษาซี เนื้อหา และตัวอย่างที่ปรากฏในหนังสือคู่มือนี้ นำมาจากต้นฉบับโดยไม่มีการดัดแปลงแก้ไขแต่อย่างใด (ยกเว้นบางตัวอย่างซึ่งได้เรียบเรียงใหม่เพื่อให้เข้าใจง่ายขึ้น) หลายๆ ตัวอย่างที่ยกมาประกอบคำอธิบายโดยผู้แต่งทั้ง 11 ท่านนั้น จัดว่าเข้าใจยาก สำหรับผู้เริ่มต้นเขียนภาษาซีใหม่ๆ ที่จะเข้าใจได้ทันที ทำให้ต้องคิดกลับไปมาครั้งหลายครั้ง ถึงจะเข้าใจได้อย่างถ่องแท้ บ่อยครั้งการนึกหาตัวอย่างนอกเหนือไปจากที่มีในต้นฉบับ เพื่อทำความเข้าใจกับคำอธิบายที่ค่อนข้างสั้นและกระชับ ไม่ต่างไปจากการงมเข็มในมหาสมุทร เช่น เหตุผลของการใส่วงเล็บเพิ่มอีกชุดใน macro

```
#define         product(a, b)           ((a) * (b))
```

แทนที่จะเขียนผิดๆเป็นเพียง

```
#define         product(a, b)           (a) * (b)
```

เพื่อให้ทำงานถูกต้องตรงกับความต้องการ ดูจะพื้นๆ ไม่มีลับลมคมใน แต่เมื่อลองนึกถึงตัวอย่างที่จะนำไปใช้จริง ทั้งกรณีที่ถูกและผิด กลับไม่ใช่ของง่ายนัก ประสพการณ์ และความมานะ บากบั่นเท่านั้นที่จะช่วยนำไปสู่ความสำเร็จ ดั่งเช่นผู้แต่งทั้ง 11 ท่าน ที่ได้รวบรวมมาให้เราได้ใช้เป็นแนวทางในการปฏิบัติ เพื่อการเขียนโปรแกรมภาษาซีที่มีมาตรฐาน และถูกต้องในการใช้งานตลอดไป

พีระพนธ์ โสพัศสถิตย์

1 พฤษภาคม 2540

# บทนำ

เป็นที่ยอมรับกันว่า ภาษา C เป็นภาษาที่ได้รับความนิยมไม่ว่าจะเป็นวงการศึกษา หรืออุตสาหกรรมซอฟต์แวร์ที่พัฒนาระบบซอฟต์แวร์ และแอปพลิเคชันต่างๆ ความนิยมดังกล่าวก่อให้เกิดการขยายตัวของการเรียนรู้ ถ่ายทอด ทดลอง และปฏิบัติเกี่ยวกับการเขียนโปรแกรมภาษา C กันอย่างแพร่หลาย [1, 6, 8] จากโปรแกรมคลาสสิก "hello world" ของ K&R เป็นแอปพลิเคชันที่สลับซับซ้อน วิธีการพัฒนาและเขียนโปรแกรมก็เริ่มแตกต่างกันออกไป โปรแกรมเมอร์แต่ละคนจะมีความชอบและถนัดในสไตล์การเขียนโปรแกรมของตนเอง เมื่อมีคนเขียนโปรแกรมภาษา C มากขึ้น รูปแบบก็หลากหลาย ลักษณะสำคัญอีกประการหนึ่งของภาษา C ซึ่งจะมองข้ามไม่ได้คือ ความกระชับของ syntax ในภาษาที่สามารถเขียนโปรแกรมให้สั้น (สุดๆ) แต่สามารถทำงานในสิ่งที่โปรแกรมเมอร์ต้องการได้เหมือนกับภาษาอื่น ซึ่งอาจจะต้องใช้ความยาวมากกว่าหลายเท่า ในการปฏิบัติงานเดียวกัน เหตุผลดังกล่าวทำให้โปรแกรมภาษา C อ่านยาก ผู้ที่เคยตรวจแก้ (debug) โปรแกรมภาษา C ของโปรแกรมเมอร์อื่นจะเข้าใจถึงความลำบาก และหัวเสียของการตีความแต่ละบรรทัดที่โปรแกรมเมอร์ผู้นั้นเขียน ด้วยเหตุนี้การกำหนดมาตรฐานรูปแบบการเขียนโปรแกรมจึงเป็นสิ่งจำเป็นที่จะช่วยลดความลำบาก และเวลาที่ใช้ในการอ่านตีความ เพื่อทำความเข้าใจกับโปรแกรมภาษา C

หลายท่านคงจะแย้งว่าโปรแกรมภาษา C ทั่วๆ ไปที่เห็นในหนังสือก็ดี จากซอฟต์แวร์ก็ดี หรือจากแหล่งข้อมูลต่างๆ เช่น อินเตอร์เน็ต (Internet) ก็ดี ทั้งหมดก็อ่านง่าย ดูเข้าใจไม่ยาก ทุกคนก็ดูเหมือนจะเขียนเหมือนๆ กันหมด ลองมาดูรายละเอียดของการเขียนโปรแกรมกัน

เริ่มจากโปรแกรมคลาสสิก "hello world"

```
main()
{
      printf("hello world\n");
}
```

ทุกคนจำโปรแกรมนี้ได้ดี (มีคำกล่าวในหมู่ผู้ใช้ภาษา C ว่า คนที่ไม่รู้จักโปรแกรมนี้ไม่ควรจะเรียกตัวเองว่าเป็นโปรแกรมเมอร์ภาษา C) เพราะเป็นตัวอย่างแรกของหนังสือภาษา C ที่แต่งโดยผู้สร้างภาษา 2 คนคือ Dennis M. Ritchie และ Brian W. Kernighan หรือที่รู้จักกันโดยย่อว่า K&R จาก K&R จนถึงปัจจุบัน ภาษา C ผ่านขั้นตอนวิวัฒนาการจนกลายมาเป็น ANSI C ซึ่งเป็นมาตรฐานสากลของภาษา C ที่ใช้กันทุกวันนี้ โปรแกรมข้างต้นจึงต้องเขียนใหม่เพื่อให้สอดคล้องกับมาตรฐาน ANSI C ดังนี้

```
int
main(void)
{
      printf("hello world\n");
      return 0;
}
```

เพราะ ANSI C กำหนดว่า main ในภาษา C จะต้อง return ค่า ***int*** เสมอ เพียงแค่นี้ก็จะเห็นว่าโปรแกรมภาษา C ที่เห็นๆ กันก็ผิดไปจากมาตรฐานกว่าครึ่ง เพราะโปรแกรมเหล่านั้นมักจะใช้ ***void*** แทน int ซึ่งเป็นค่าที่ควรจะ return โดย main ที่กล่าวข้างต้นเป็นเพียงตัวอย่างง่ายๆ โดยชี้ให้เห็นว่า วิวัฒนาการ และความนิยมของภาษา ได้เพิ่มความสลับซับซ้อนให้กับโปรแกรมภาษา C มาก สิ่งที่โปรแกรมเมอร์ทุกคนควรจะคำนึงถึงก็คือ "เขียนโปรแกรมให้อ่านง่าย

ที่สุด" ซึ่งดูจะพูดง่ายแต่ทำยาก จากประสบการณ์ของผู้เขียนเอง ในระยะแรก ผู้หัดใหม่ๆ มักจะเขียนโปรแกรมแบบง่ายๆ ตรงๆ จากความคิดของตนเอง พอเริ่มจะเดินได้ ยังไม่ทันจะมั่นคงดี ก็ออกวิ่งทันที เมื่อปีกกล้าขาแข็งขึ้นก็เขียนโปรแกรมแบบแผลงๆ ที่อ่านรู้เรื่องคนเดียว (6 เดือนให้หลังตัวคนเขียนเองก็อาจจะอ่านไม่เข้าใจเหมือนกัน) เช่น

```
for (u=q=p->st.parm; q && u&(1<<27) || p->st.next; p++, q++, u%=q)
    { do_something; }
```

ด้วยเหตุนี้สิ่งแรกที่ควรปลูกฝังให้กับผู้เขียนโปรแกรมภาษา C คือ "**โปรแกรมเมอร์ที่เก่ง (ดี) คือ คนที่เขียนโปรแกรมแล้วคนอื่นอ่านรู้เรื่อง เข้าใจได้โดยง่าย**"

# 1. Introduction

โดยเหตุที่ ANSI ไม่ได้กำหนดรูปแบบการเขียนโปรแกรมอย่างเป็นทางการ จึงไม่มีข้อกำหนดกฎเกณฑ์อะไรว่า การเขียนโปรแกรมภาษา C แบบไหนจึงจะถูกต้องตามหลักการ 100 เปอร์เซ็นต์ มาตรฐานรูปแบบของโปรแกรมในภาษา C ที่จะกล่าวถึงในที่นี้ เป็นมาตรฐานที่เรียกว่า Indian Hill Style ซึ่งเป็นมาตรฐานที่ได้รับความนิยมอย่างแพร่หลายในหมู่ผู้พัฒนาโปรแกรมภาษา C มาตรฐานดังกล่าวกำหนดโดย Bell Labs และผู้ทรงคุณวุฒิในคณะอนุกรรมการของ ANSI C อีก 3 ท่าน เราจะมาดูรายละเอียดของหลักการแต่ละขั้นของมาตรฐานดังกล่าว

"*To be clear is professional; not to be clear is unprofessional*."—Sir Ernest Gowers.

# 2. File Organization

ขอเริ่มจากการจัดการเกี่ยวกับ file ซึ่งประกอบด้วยส่วนต่างๆ แยกจากกันด้วยการเว้นบรรทัด แม้ว่าจะไม่มีการกำหนดขนาดใหญ่ที่สุดที่ควรจะเป็น file หนึ่งๆ ไม่ควรยาวเกิน 1000 บรรทัด เพราะทั้ง editor และ compiler จะทำงานลำบาก การใส่เครื่องหมายดอกจันทน์เป็นตัวคั่นระหว่างบรรทัด ทำให้เสียเวลาในการ scroll จึงไม่แนะนำให้ใช้ บรรทัดใดที่ยาวเกิน 79 ตัวอักษร (columns) ไม่แนะนำ เพราะปัญหาเกี่ยวกับการแสดงผลของจอ บรรทัดที่ยาวมากๆ อันเนื่องมาจากการย่อหน้า (indent) หลายระดับ บอกถึงการจัดรูปแบบโปรแกรมที่ไม่ดี

## 2.1 File Naming Conventions

ชื่อ file ประกอบด้วยชื่อ (base name) และสกุล (คั่นด้วยจุด fullstop) อักษรตัวแรกของชื่อควรจะเป็นตัวอักษร ควรใช้อักษรตัวเล็ก (lowercase) กับตัวอักษรทุกตัวที่เป็นชื่อ file (ยกเว้นจุด fullstop) อาจจะมีตัวเลขเป็นส่วนหนึ่งของชื่อได้ ชื่อไม่ควรยาวเกิน 8 ตัวอักษร สกุลไม่ควรเกิน 3 ตัวอักษร (รวมจุด fullstop เป็นสี่) กฎนี้ใช้กับ program files และ default files ที่สร้างขึ้นโดยโปรแกรม (เช่น "rogue.sav") และเพื่อความสะดวกในการ port โปรแกรมไปบนเครื่องรุ่นเก่า (upward compatibility).

Compilers และ tools บางตัว บังคับให้ใช้กฎเกณฑ์ตามที่กำหนด [5] สกุลต่อไปนี้เป็นกฎที่พึงปฏิบัติ

1. โปรแกรมภาษาซี ต้องมีสกุล .c
2. โปรแกรมภาษา assembly ต้องมีสกุล .s

ข้อกำหนดต่อไปนี้ เป็นข้อกำหนดมาตรฐานสากล

1. Relocatable object file ต้องมีสกุล .o

2. Include header file ต้องมีสกุล .h รูปแบบอีกประเภทหนึ่งที่เป็นที่นิยมกันมากใน multi-language environment คือการใช้ชนิดของภาษาเป็นส่วนหนึ่งของสกุล เช่น foo.c.h หรือ foo.ch
3. Yacc ต้องมีสกุล .y
4. Lex ต้องมีสกุล .i

ส่วน C++ ขึ้นอยู่กับข้อกำหนดของ compiler นั้นๆ ซึ่งประกอบด้วย .c, ..c, .cc, c.c, และ .C เนื่องจากโปรแกรมภาษาซีสามารถแปลโดย compiler ของ C++ จึงไม่มีข้อกำหนดตายตัวที่แน่นอน

นอกจากนี้ ผู้ใช้ส่วนมากนิยมเขียน “Makefile” (ไม่ใช่ “makefile”) เพื่อบอกให้ ***make*** ทำงาน (เหมาะสำหรับระบบที่สนับสนุน make) และ “README” ซึ่งมักจะรวบรวมสิ่งที่เก็บอยู่ใน directory หรือ directory tree นั้นๆ

## 2.2 Program Files

ลำดับของการเรียงส่วนต่างๆ ของโปรแกรมพอจะสรุปได้ดังนี้

1. ส่วนแรกคือคำนำซึ่งบอกให้รู้ว่า file ประกอบด้วยอะไรบ้าง คำอธิบายถึงจุดประสงค์ของแต่ละ object ที่ใช้ (ไม่ว่าจะเป็น functions, external data declarations/definitions, และอื่นๆ) ทั้งนี้ อาจใส่ชื่อผู้เขียนโปรแกรม ข้อมูลสำหรับ revision control และข้อมูลอ้างอิงต่างๆ
2. ส่วนถัดมาคือ header files ที่ใช้ในโปรแกรม ถ้าการ include มีเหตุผลเฉพาะ ควรจะบันทึกเหตุผลหรือความจำเป็นดังกล่าวด้วย ในกรณีทั่วไป system include files เช่น ***stdio.h*** ควรจะมาก่อน user include files
3. ถัดมาคือ define และ typedef ที่ใช้ตลอดทั้งโปรแกรม ลำดับที่เหมาะสมคือ constant macros, function macros, typedef, และ enum
4. ส่วนต่อมาคือ global (external) data declarations ตามลำดับดังนี้ externs, non-static globals, static globals ถ้ามี define ที่เกี่ยวข้องกับ global data ส่วนใดส่วนหนึ่ง (เช่น flag word) ควรจะเรียงต่อจาก data declaration หรือใส่ไว้ใน embedded structure declarations ที่ประสงค์จะให้ define นั้นๆ ต่ำกว่า keyword ที่ประกาศใช้อีก level หนึ่ง
5. ส่วนสุดท้ายคือ functions ที่มีการจัดลำดับอย่างเหมาะสม เช่น บางกลุ่มควรจะจัดให้อยู่ด้วยกันในลักษณะ “breadth-first” จะดีกว่า “depth-first ทั้งนี้ขึ้นอยู่กับดุลพินิจของผู้ออกแบบ ถ้าเป็นการกำหนด utility functions ขนาดใหญ่ที่ไม่ขึ้นต่อกัน ควรเรียงตามลำดับอักษร

## 2.3 Header Files

เป็นข้อมูลที่ใส่เข้ามาในไฟล์ที่เรียกใช้ก่อนที่จะถูก compile โดย C preprocessor เช่น ***stdio.h*** ซึ่งจะถูกเรียกใช้โดยโปรแกรมที่ต้องติดต่อกับ standard I/O library นอกจากนี้ header files ยังประกอบด้วยข้อมูลต่างๆ ที่จำเป็นสำหรับโปรแกรม (ที่มีมากกว่า 1 ไฟล์) โดยที่ข้อมูลเหล่านั้น เป็นข้อมูลกลางที่ เรียกใช้โดยไฟล์ต่างๆ ภายในโปรแกรม เช่น macros, defines, data declarations, function prototypes, ฯลฯ การจัดแบ่งกลุ่มข้อมูลใน header file จึงควรยึดถือภาระหน้าที่ของแต่ละกลุ่มเป็นหลัก จุดประสงค์ที่สำคัญคือ ความง่ายของการ port โปรแกรมจาก

เครื่องหนึ่งไปอีกเครื่องหนึ่ง โดยข้อมูลกลางจะถูกเปลี่ยนเพื่อให้เหมาะสมกับสภาวะแวดล้อมของเครื่องใหม่ได้ง่าย เพียงแค่แก้ใน header file เท่านั้น

การตั้งชื่อ header file พยายามใช้ชื่อมาตรฐานและเก็บรวบรวมไฟล์ทั้งหมดในที่เดียวกัน ชื่อมาตรฐานคือชื่อที่ล้อมรอบด้วยสัญลักษณ์ <*name*> ถ้าเป็น header file ที่ผู้เขียนสร้างขึ้นเองจะใช้สัญลักษณ์ "..." โดยที่ไฟล์ทั้งหมดจะเก็บไว้ใน directory ปัจจุบัน (ในกรณีที่ไฟล์สร้างเองไม่ได้เก็บใน directory ปัจจุบัน จะใช้ตัวเลือก -I ในการบอก directory นั้นๆ)

ข้อควรระวังในการเขียน header file มีหลายประการคือ

1. Functions และ external variables ใน header file ใดๆ ควรจะใส่ไว้ในไฟล์ที่เรียกใช้ functions และ external variables เหล่านั้น

2. การประกาศใช้ variables ใดๆ ใน header file เป็นวิธีการเขียนโปรแกรมที่ไม่ เหมาะสม เพราะ variables บางประเภท เช่น typedef และ initialization จะถูกเรียกจากไฟล์ที่มี header file นั้นๆ มากกว่า 1 ครั้งไม่ได้

3. ไม่ควรใช้ header file ซ้อนกัน ถ้าหลีกเลี่ยงไม่ได้จริงๆ เช่น ในโปรแกรมขนาดใหญ่ที่ต้องเรียก header files หลายๆ อันจาก files ต่างๆ ในโปรแกรม อาจจะรวมเอา ชื่อ header files เหล่านั้นไว้ด้วยกันใน header file เดียว แล้วเรียก header file ร่วม อันนี้ก็เพียงพอ

4. ควรใส่ตัวอย่าง header ต่อไปนี้ไว้ใน header file ทุกๆ อัน เพื่อป้องกันการเรียกซ้ำซ้อน (double inclusion) โดยเฉพาะในกรณีที่มี nested includes ดังตัวอย่าง

```
#ifndef     EXAMPLE_H
#define     EXAMPLE_H
        ...   /* body of example.h file     */
#endif      /* EXAMPLE_H                    */
```

ลองดูตัวอย่าง header file และ #define

```
/*
 *   This file contains two functions, namely, proc_student and
 *   do_report.  Each function, in turn, calls a number of header
 *   files included in a user-defined header allfile.h."
 *   The following...
 */
#include        "allfile.h"
#include        "local_n.h"
#define         SIZE        40
#define         LENGTH      80
typedef         struct      student
{
      char      f_name[SIZE];
      char      L_name[LENGTH];
      double    GPA;
      int       credit;
} STUDENT_REC;
int             max_student;            /* global variable    */
void            proc_student(void);     /* function prototype */
void            do_report(STUDENT_REC);
```

โดยที่ allfile.h ประกอบด้วย

```
#include    <stdio.h>
#include    <stdlib.h>
#include    <string.h>
#include    <math.h>
#ifndef     MACHINE
#define     MACHINE
      /* body of machine dependent features */
#endif
#define     MAX(i,j)    ((i)>(j)) ? (i) : (j)
```

## 2.4 Other Files

โดยทั่วไป มักจะมี file ชื่อ “README” ที่กล่าวถึงภาพรวมของโปรแกรม และสาระสำคัญๆ เช่น การใส่รายการ conditional compilation flags ทั้งหมด รวมทั้งความหมายของแต่ละรายการ และรายการของ file ที่เกี่ยวกับสิ่งที่เป็น machine dependent

# 3. Comments

“*When the code and the comments disagree, both are probably wrong*.”—Norm Schryer

เรื่องต่อไปที่จะพูดถึง ซึ่งทุกคนมักจะมองข้าม หรือไม่ให้ความสนใจเท่าที่ควรคือ comment การใส่ comment ในโปรแกรมซึ่งเป็นส่วนหนึ่งของ documentation เป็นสิ่งที่ต้องทำ ไม่ใช่ควรทำ เพราะ comment จะบอกว่า โปรแกรมทำอะไร ทำอย่างไร มี parameters อะไรที่เกี่ยวข้อง และถูกเปลี่ยนแปลงค่าอย่างไร ข้อจำกัดของแต่ละ block มีอะไรบ้าง เช่น while จะทำงานจนกว่าจะถึง EOF หรือ จำนวน records มากกว่า MAX_NO เป็นต้น ควรระวังการ comment ที่เกินความจำเป็น เช่น

```
      y = x + 2;    /* Add 2 to x and store the sum in y */
```

โดยทั่วไปแล้ว การใส่ comment ที่ต้นโปรแกรม (หรือ file) จะดีกว่าการใส่ comment บรรทัดต่อบรรทัดแบบโปรแกรมภาษา assembly การเขียน comment ก็มีกติกาเหมือนกัน ดังตัวอย่างวิธีการจัดวาง comment ต่อไปนี้

```
/*
 * This is a block of comment
 */

/*
** Alternate format for block of comments
*/
```

เหตุที่ใช้วิธีดังกล่าวก็เพื่ออำนวยความสะดวกในการค้นหา comment ภายในโปรแกรม โดยใช้คำสั่งง่ายๆ เช่น grep “^.*” บน UNIX ก็สามารถเรียก comment ดูได้ทั้งหมด เป็นการช่วยในการทำ documentation อีกด้วย

ในกรณีที่ statement หรือ block ใดๆ มีความหมายพิเศษที่ต้องการจะอธิบายหรือชี้เฉพาะ วิธีการใส่ comment ก็ทำได้ 2 แบบคือ

```
if (argc > 1)
{
     /* get input file from command line */
     if (freopen(argv[1], "r", stdin) == NULL)
     {
          perror(argv[1]);
     }
}
```

หรือ

```
if (value == OVERFLOW)
{
     flag = TRUE;                    /* special case      */
}
else
{
     flag = regular(value);          /* only non-overflow */
}
```

## 4. <u>การประกาศตัวแปร</u> (Declarations)

ตัวแปรประเภท global ควรจะเริ่มต้นที่ column 1 ตัวแปรชนิด external ต้องเริ่มต้นด้วย keyword ***extern*** ถ้าตัวแปรชนิด extern เป็น array ที่มีขนาดแน่นอน ต้องกำหนดขนาดของ external array นั้นทุกครั้งไป นอกเสียจะกว่าขนาดนั้นถูกกำหนดไว้เรียบร้อยแล้วในตัวมันเอง (เช่น read-only character array ที่เป็น null-terminated) การกำหนดขนาดไว้ทุกครั้งเช่นนี้ เป็นประโยชน์ต่อผู้ที่อ่านโปรแกรมที่คนอื่นเขียนได้ง่ายขึ้น

ตัวกำกับชนิดตัวแปรประเภทหนึ่ง คือ pointer (สัญลักษณ์ `*') ควรจะเขียนติดกับชื่อตัวแปร ไม่ใช่ประเภท (type) ของตัวแปลที่หลายๆ คนทำกัน กล่าวคือ

```
char *s, *t;
```

ไม่ใช่เป็น

```
char* s, t, u;
```

เพราะแบบหลังนั้นผิด เนื่องจากว่า t และ u ไม่ได้เป็น char pointers

การประกาศตัวแปรที่ไม่เกี่ยวข้องกัน แม้ว่าจะเป็นตัวแปรชนิดเดียวกัน ควรจะประกาศแยกกันคนละบรรทัด โดยมี comment กำกับหน้าที่ของ object แต่ละตัวที่ประกาศ ยกเว้น #define เพราะส่วนมากชื่อก็พอจะสื่อความหมายให้เข้าใจว่า constant แต่ละตัวคืออะไร ชื่อตัวแปร ค่าของตัวแปรและ comment ควรจะจัดเรียงให้ตรงกัน (ควรใช้ tab แทน blank หรือ space) เช่น

```
/* defines for string    */
#define    SIZE       80
#define    OCCUR      10
#define    MAX_S      100
```

```
/* defines for boat.type */
#define    KETCH      (1)
#define    YAWL       (2)
#define    SLOOP      (3)
#define    SQRIG      (4)
#define    MOTOR      (5)
```

สำหรับ structure และ union template ที่ใช้ในการประกาศรายละเอียดควรจะจัด ให้แต่ละ element อยู่คนละบรรทัด โดยมี comment กำกับ วงเล็บปีกาควรจะวางในตำแหน่ง ที่เหมาะสม (จะใช้ K&R style หรือ ไว้ที่ column 1 ก็ได้ แต่ต้องให้เหมือนกันหมดทั้งโปรแกรม) เช่น

```
struct    mydef
{
     int       length;         /* column 1       */
     int       occurs;         /* Size of struct */
     char      code;           /* data code      */
};
```

#define ข้างต้นมักจะวางไว้ใต้ struct ที่เกี่ยวข้อง ในกรณีนี้คือ struct mydef หรือจะใส่ไว้ภายใน {...} ของ struct mydef ก็ได้ ในบางกรณี *enum* อาจจะมีประโยชน์กว่า #define ก็ได้ เช่น

```
enum           bt { KETCH=1, YAWL, SLOOP, SQRIG, MOTOR };
struct         boat
{
     int       wllength;       /* water line length in meters */
     enum bt   type;           /* what kind of boat           */
     long      sailarea;       /* sail area in square mm      */
};


enum           ct {SECOND = 1, MIN, HOUR};
struct         mydef
{
     int       length;         /* size of struct     */
     enum ct   type;           /* freq of occurrence */
     long      code;           /* data code          */
};
```

ถ้าค่าตัวแปรใดที่จำเป็นต้อง initialize เพราะค่าเริ่มต้นมีความสำคัญมาก จะต้อง initialize ต่างหาก หรืออย่างน้อยที่สุดก็ต้องใส่ comment เพื่อบอกว่าค่า default initialization (ใน compiler บางตัวจะ set เป็นศูนย์) ถูกนำมาใช้เป็นค่าเริ่มต้นในกรณีดังกล่าว ข้อห้ามก็คือ struct ที่มี { } เปล่าๆ ไม่ควรจะใช้ ค่าที่ใช้ในการ initialize structure ควรจะจัดเป็นชุดให้ตรงกับ definition ถ้าค่าใดที่เป็น constant ชนิด long ก็ควรจะประกาศว่าเป็น long โดยใช้ L (ตัวใหญ่) เพราะถ้าใช้ตัวเล็กแล้ว “2l” จะมองดูเหมือน “21” (ยี่สิบเอ็ด)

```
int       x = 5;
char      *msg = “hello world”;
struct    mydef array[] =
{
     {20, MIN,    6000000L},
     {8,  SECOND, 0L},
     {0},
};
```

ในการเก็บค่าตัวแปรที่ประกาศใช้ในโปรแกรมลงใน file ซึ่งเป็นส่วนหนึ่งของโปรแกรม ขนาดใหญ่ (ประกอบด้วยหลาย files), function และตัวแปรที่ใช้มากควรจะประกาศเป็น ***static*** เพื่อให้ scope นั้นอยู่แค่ local file ตัวแปรที่ต้อง share กับ file อื่นๆ ควรจะประกาศเป็น extern และให้มีจำนวนน้อยๆ ต้องใส่ comment กำกับใช้ชัดเจนว่าใช้ร่วมกับ file อื่น โดยใน comment นั้นควรจะใส่ชื่อ file อื่นๆ ด้วย (เพื่อสะดวกในการ cross reference) ถ้า debugger ที่ใช้ไม่แสดงค่า static variables/objects ในขณะที่ทำการ debug ให้สร้างนิยามใหม่คือ STATIC โดยใช้ ***#define STATIC*** เป็นค่าอะไรก็ได้ตามต้องการ ระหว่างการ debug

ชนิดของตัวแปรที่มีความสำคัญที่สุด ควรจะ highlight ด้วยการใช้ typedef แม้ว่าค่านั้นจะเป็นแค่ integer เพราะชื่อที่ unique เหล่านี้ทำให้โปรแกรมอ่านง่าย (โดยที่มี typedef ของ integer เป็นจำนวนไม่มาก) structure อาจจะทำเป็น typedef ก็ได้เช่นกันถ้าต้องการเน้น โดยมากจะนิยมใช้ชื่อเดียวกัน เช่น

```
typedef     struct    something
{
     int    first;
     char   *name, *sp_alias;
} something;
```

ชนิดของค่าที่ return โดย function หนึ่งๆ ควรจะประกาศให้ชัดเจน (แม้ว่าจะเป็น int ก็ตาม) ถ้า compiler supports prototype ก็เขียน prototype ด้วย ข้อผิดพลาดที่พบบ่อยๆ ก็คือ การละเลยการประกาศชนิดของค่าที่ return จาก external math functions ซึ่งมักจะเป็น double Compiler ส่วนมากจะอนุมานว่า function ดังกล่าว return ค่า int (ในกรณีที่ไม่ประกาศว่าเป็น double) ซึ่งทำให้ค่าที่ return ถูก convert เป็นค่าของตัวแปรแบบ floating ที่ผิดความหมาย

"*C takes the point of view that the programmer is always right.*"—Michael DeCorte

## 5. Function Declarations

ในการเขียน function ควรเริ่มต้นด้วย comment ซึ่งเป็นบทนำในการกล่าวถึงหน้าที่ของ function นั้น และวิธีการ (เรียก) ใช้ อาจจะรวมถึงการตัดสินใจเกี่ยวกับหลักการออกแบบที่สำคัญๆ และผลกระทบข้างเคียง (side effects) ของ function นั้น ไม่ต้องกล่าวซ้ำถึงสิ่งที่มองเห็นได้ชัดจาก code ของ function

ค่าที่ function ส่งคืนควรจะใส่ไว้เป็นบรรทัดเดียวต่างหาก ซึ่งอาจจะย่อหน้า 1 tabstop ก็ได้ อย่าปล่อยให้ default เป็นค่า ***int*** ถ้า function ไม่ได้ return ค่าอะไรให้ใช้ ***void*** (สำหรับ compiler ที่ไม่ support void ก็ให้ใช้ #define void หรือ #define void int แทนค่า void) ถ้าค่าที่ส่งคืนต้องมีคำอธิบายเพิ่ม เพื่อความกระจ่างควรจะอธิบายใน comment ข้างต้น ถ้าไม่ยาวมากก็อาจจะต่อท้าย (บนบรรทัดเดียวกัน) ค่า return ก็ได้ โดยเว้นวรรคสัก 1 tabstop ชื่อ function และ formal parameter list ควรจะอยู่บนบรรทัดเดียวโดดๆ โดยเริ่มที่ column 1 local variable และ code ของ function ควรจะย่อหน้า 1 tabstop วงเล็บปีกกาเปิดและปิดควรอยู่บนบรรทัดเดียวโดดๆ และเริ่มที่ column 1

โดยทั่วไป หน้าที่ของ variable แต่ละตัวควรจะอธิบายให้ชัดเจน ซึ่งอาจจะกระทำได้โดยใส่ใน comment ข้างต้น หรือต่อท้ายบนบรรทัดเดียวกับ variable (วิธีหลังไม่แนะนำ) loop counter "i", pointers ประเภท strings "s", และ int ที่ใช้เป็น character "c" มักจะไม่ต้องอธิบาย (ตามความนิยม) ถ้ากลุ่มของ function ใดที่มี parameter

เหมือนๆ กัน ควรใช้ชื่อเหมือนกันใน function กลุ่มนั้น และวางในตำแหน่งเดียวกัน โดยนัยกลับกัน parameters ที่ต่างกันก็ใช้คนละชื่อเพื่อไม่ให้ซ้ำกัน

comment ภายใน function ควรจะย่อหน้าให้ตรงกับ code และคั่นด้วยบรรทัดว่าง ถ้าเป็น comment ของ code คนละส่วน

ควรระวังการใช้ function ที่มี arguments ไม่จำกัด-จำนวน เช่น varargs เพราะวิธีการดังกล่าวไม่ portable 100% ควรจะออกแบบ interface ที่ใช้ parameters เป็นจำนวนที่แน่นอน (fixed) ถ้าหลีกเลี่ยงไม่ได้จริงๆ ใช้ macros จาก library ในการประกาศ arguments ของ functions ที่มี arguments ได้ไม่จำกัดจำนวน

ถ้าใน function มี external variables หรือ functions ที่ไม่ได้ประกาศในส่วน global ของ file จะต้องประกาศชื่อเหล่านั้นโดยนำหน้าด้วย keyword ***extern***

หลีกเลี่ยงการใช้ชื่อ local variables ที่ซ้ำกับ parameters ระดับที่สูงกว่า เพราะ local จะ override นิยามของ parameters ที่สูงกว่า โดยเฉพาะอย่างยิ่งไม่ควรใช้ชื่อซ้ำใน nested loops หรือ blocks แม้ว่าจะไม่ผิดตามกฎเกณฑ์ของภาษา แต่เป็นจุดที่ทำให้เกิดความสับสนได้ง่าย (โดยเฉพาะถ้าใช้ ***lint*** ด้วย option -h)

## 6. Whitespace

```
int i;main(){for(;i["]<i;++i){--i;}"];read('-'-'-',i+++"hell\
 o, world!\n",'/'/'/'));}read(j,i,p){write(j/p+p,i---j,i/i);}
```

- Dishonorable mention, Obfuscated C Code Contest, 1984.

Author requested anonymity.

พยายามใช้ whitespace ทั้งในแนวนอนและแนวตั้ง การย่อหน้าควรจะแสดงให้เห็นโครงสร้างของแต่ละ block อย่างเด่นชัด เช่น เว้นวรรคอย่างน้อย 2 บรรทัด เมื่อจบ function หนึ่ง ก่อนที่จะเริ่ม comment ของ function ถัดไป

ในกรณีที่ต้องใช้เงื่อนไขยาวๆ ให้แยกบรรทัดกันโดยเว้นวรรคและย่อหน้าให้เหมาะสม เช่น

```
if(foo->next==NULL&&totalcount<needed&&needed<=MAX_ALLOT&&
server_active(current_input))
{
      ...
}
```

```
if (foo -> next == NULL
            && totalcount < needed
            && needed <= MAX_ALLOT
            && server_active(current_input))
{
      ...
}
```

ในทำนองเดียวกัน สำหรับ for loops ที่ซับซ้อนมากๆ

```
for (curr = *listp, trail = listp;
            curr != NULL;
            trail = &(curr->next), curr = curr->next)
{
      ...
}
```

สำหรับ expression ที่ซับซ้อนประเภทอื่นก็ควรทำเช่นเดียวกัน

```
c = (a == b)
      ? d + f(a)
      : f(b) - d;
```

Keyword ที่ตามด้วย expression ในวงเล็บ ควรจะวางห่างจากวงเล็บเปิด 1 ช่องไฟ (ยกเว้น *sizeof*) ควรใส่ blank 1 ช่องหลัง comma ที่คั่น argument lists เพื่อให้อ่านง่าย แต่สำหรับ argument lists ของ macros ไม่ควรใส่ blank ระหว่างชื่อกับวงเล็บเปิด มิฉะนั้น C preprocesssor จะไม่รู้จัก argument lists เลย (เพราะคิดว่าวงเล็บเปิดเป็นส่วนหนึ่งของนิยาม - ข้อควรระวังอย่างยิ่ง)

## 7. Examples

มาดูตัวอย่างง่ายๆ กันสักตัวอย่าง

```
/* Determine if the sky is blue by checking that it isn't
 * night.
 * CAVEAT: Only sometimes right.  May return TRUE when the
 * answer is FALSE.  Consider clouds, eclipses, short days.
 * NOTE: Uses 'hour' from 'hightime.c'.  Returns 'int' for
 * compatibility with the old version.
 */

int                                 /* true or false           */
skyblue()
{
      extern int  hour;             /* current hour of the day */

      return hour >= MORNING && hour <= EVENING;
}

/* Find the last element in the linked list pointed to by
 * nodep and return a pointer to it.  Return NULL if there
 * is no last element.
 */
```

```
node_t *
tail(nodep)
node_t      *nodep;                         /* pointer to head of list */
{
      register    node_t      *np;          /* advances to NULL        */
      register    node_t      *lp;          /* follows one behind np   */

      if (nodep == NULL)
            return NULL;
      for (np = lp = nodep; np != NULL; lp = np, np = np->next)
            ;                               /* VOID                    */
      return lp;
}
```

ตัวอย่างข้างต้นเป็น C แบบเก่า คือ K&R C ซึ่งผู้อ่านหลายท่านคงจะไม่เคยเห็นมาก่อน อันที่จริงก็ไม่ต่างจาก ANSI C มากนัก แต่ในที่นี้เราจะพูดถึง style ของการเขียนภาษา C เท่านั้น

ในแง่ของ format ตัวอย่างนี้ถือเป็นมาตรฐานที่พึงปฏิบัติอย่างยิ่ง โดยเฉพาะ for ที่ไม่มี statement ในส่วนของ loop body เครื่องหมาย ";" จะต้องวางไว้บนบรรทัดเดียวโดดๆ และควรจะมี comment ดังตัวอย่าง ห้ามพิมพ์เครื่องหมาย ";" หลังวงเล็บปิดของ for เด็ดขาด เพราะคนอ่านจะมองข้ามได้ง่าย และทำให้เกิดความสับสน ข้อเสียของ for นี้ก็คือ expression ในส่วนของ loop ซับซ้อนเกินไป ควรจะแตกเป็น

```
for (np = lp = nodep; np != NULL; np = np->next)
      lp = np;
```

## 8. Simple Statements

การเขียนคำสั่งในภาษาซี จะแยกเป็นคำสั่งๆ หรือที่เรียกว่าประโยค (statement) ซึ่งจะจบด้วย ";" ที่ท้ายประโยคเสมอ ประโยคในภาษาซีแบ่งออกเป็น 2 ชนิด คือ simple statements และ compound statements

ในการเขียน simple statement นั้น ควรจะจัดให้มีบรรทัดละ 1 ประโยคเท่านั้น ยกเว้นว่าแต่ละประโยคมีส่วนเกี่ยวข้องกันอย่างใกล้ชิด เช่น

```
case FOO: oogle(zork); boogle(zork); break;
case BAR: oogle(bork); boogle(zork); break;
case BAZ: oogle(gork); boogle(bork); break;
```

ตัวอย่างประโยคพิเศษที่ควรจะเขียนอยู่บนบรรทัดเดียวโดดๆ คือ null body ของ ***for*** หรือ ***while*** loop เพื่อแสดงให้ผู้อ่านเห็นว่าเป็นความตั้งใจของโปรแกรมเมอร์ ไม่ใช่ลืม เช่น

```
while (*dest++ = *src++)
      ;                         /* VOID          */
```

Style ที่ดีควรใส่ comment ไว้ข้างท้ายด้วย (ดังตัวอย่างข้างต้น)

การทดสอบที่ไม่เท่ากับศูนย์ อย่าใช้วิธี default คือ ควรจะเขียนในรูปของ

```
if (f() != FAIL)
```

จะดีกว่าที่เขียนว่า

```
if (f())
```

แม้ว่า FAIL จะถูกกำหนดให้มีค่าเป็นศูนย์ก็ตาม ซึ่งในภาษาซีถือว่าเป็นค่าไม่จริง (false) การเขียนทดสอบชัดๆ (explicit test) จะช่วยในระยะยาว กล่าวคือ เมื่อบางคนเปลี่ยนใจจะกำหนดค่าเป็น -1 แทนที่จะเป็นศูนย์ แบบแรกจะใช้ได้ตลอด การเปรียบเทียบตรงๆ เช่นนี้เป็นสิ่งที่พึงกระทำอย่างยิ่ง แม้ว่าค่าที่ใช้เปรียบเทียบจะคงที่ไม่เปลี่ยนแปลง ดังตัวอย่าง

```
if (!(bufsize % sizeof(int)))
```

ควรจะเขียนเป็น

```
if ((bufsize % sizeof(int)) == 0)
```

เพื่อเป็นการแสดงให้เห็นว่าเป็นการเปรียบเทียบค่าตัวเลข ไม่ใช่ ***boolean*** จุดที่เกิดปัญหาบ่อยคือ strcmp ที่ทดสอบว่า string เท่ากันหรือไม่ ผลลัพธ์ไม่ควรจะเขียนแบบ default (คือละเว้น 0) ความนิยมมักจะเขียนในรูปของ macro เช่น

```
#define  STREQ(a,b)  (strcmp((a), (b)) == 0)
```

การทดสอบที่ไม่เป็นศูนย์ที่เขียนเป็นแบบ default มักจะใช้กับ predicate หรือ functions, expressions ซึ่งมีคุณสมบัติตามข้อจำกัดดังต่อไปนี้

- มีค่า 0 เมื่อ false จะไม่เป็นอย่างอื่นอีก
- ตั้งชื่อในลักษณะที่เข้าใจได้ทันทีในกรณีที่ return ค่าจริง เช่น predicate ที่มักนิยมใช้กันคือ ***isvalid*** หรือ ***valid*** (อย่าใช้ ***checkvalid***)

ในทางปฏิบัติ ความนิยมอย่างหนึ่งคือการสร้างชนิดของค่า (type) “bool” เก็บไว้ใน global include file วิธีการดังกล่าวช่วยให้โปรแกรมอ่านง่ายขึ้น เช่น

```
typedef  int         bool;
#define  FALSE       0
#define  TRUE        1
```

หรือ

```
typedef  enum        { NO = 0, YES }   bool;
```

ด้วยวิธีการเช่นนี้ ก็ยังวางใจไม่ได้ในการเขียนโปรแกรมจริงๆ อย่าเปรียบเทียบค่าของ boolean เท่ากับ 1 (TRUE หรือ YES) ควรจะเปรียบเทียบว่าไม่เท่ากับ 0 (FALSE หรือ NO) เพราะฟังก์ชั่นส่วนมากจะ return ค่า 0 ถ้า false, แต่จะ return ค่า non-zero ถ้า true ดังตัวอย่าง

```
if (func() == TRUE)  {...}
```

ควรจะเขียนเป็น

```
if (func() != FALSE) {...}
```

และจะดียิ่งขึ้น (ถ้าเป็นไปได้) ในการเปลี่ยนชื่อ function/variable หรือเขียน expression ใหม่ เพื่อให้สื่อความหมายชัดเจน โดยไม่ต้องเปรียบเทียบกับ true/false เช่น เปลี่ยนชื่อเป็น ***isvalid***()

ในบางกรณี การใช้ embedded assignment เป็นวิธีที่ดี หรือเหมาะสมที่สุด โดยไม่มีทางใดที่จะเลี่ยง ซึ่งอาจจะทำให้โปรแกรมดูเทอะทะหรืออ่านยาก ถ้าไม่ใช้ เช่น

```
while ((c = getchar()) != EOF)  {...}
```

Operator ++ และ -- นับเสมือน assignment ซึ่งมักจะนำไปสู่ side effects การเขียน embedded assignment เพื่อเพิ่มประสิทธิภาพตอน run-time ควรจะคำนึงถึงข้อดีและเสียของการเพิ่มความเร็ว แต่ลด maintainability อันสืบเนื่องมาจากการใช้ embedded assignment ผิดที่ ดังตัวอย่าง

```
a = b + c;
d = a + r;
```

ไม่ควรจะรวมเป็น

```
d = (a = b + c) + r;
```

แม้ว่าวิธีหลังจะประหยัดไป 1 cycle ในระยะยาว ความแตกต่างเพียง 1 cycle จะมีผลน้อยลงเมื่อ optimizer เก่งขึ้น (คือปล่อยให้เป็นหน้าที่ของ optimizer ดีกว่า) เพราะวิธีแรก maintain ง่ายกว่าเมื่อเทียบกับความจำของมนุษย์ที่ต้องอ่าน code เยอะๆ

คำสั่ง goto ควรจะใช้เท่าที่จำเป็นภายใต้โปรแกรมที่เขียนอย่างเป็นระเบียบแบบแผน ที่ใช้บ่อยๆ คือ ในกรณีที่ต้อง break ออกจาก ***switch***, ***for*** และ ***while*** ซ้อนๆ กันหลายชั้น ซึ่งในกรณีเช่นนั้น ส่อให้เห็นถึงการเปลี่ยนแปลงหรือแก้ไข ส่วนที่อยู่ชั้นในของ loop เป็น function ที่ return ค่า success/failure เช่น

```
        for (...)
        {
              while (...)
              {
                    if (disaster)
                          goto error;
              }
        }
        ...
error:
        clean up the mess
```

ในกรณีที่มีความจำเป็นต้องใช้ ***goto*** การเขียน label ควรจะวางไว้บนบรรทัดเดียวโดดๆ และย่อหน้า statement ที่ตามมาใต้ label นั้น และ goto ดังกล่าวควรจะใส่ comment ไว้ตอนต้นของ block นั้นๆ เพื่อให้รู้ถึงการทำงานและจุดประสงค์ของ goto ส่วน ***continue*** ก็เช่นกัน ควรจะใช้เท่าที่จำเป็นและวางไว้ใกล้ๆ ต้น loop สำหรับ ***break*** จะสร้างปัญหาน้อยกว่า

Parameters ของ functions ที่ไม่มี prototype อาจจะมีการทำ promotion ให้ชัดเจน เช่น function ที่ต้องการค่า 32-bit ***long*** แล้วได้ค่า 16-bit ***int*** ส่งมาให้แทน จะเกิดการทำงานผิดพลาดขึ้นเพราะ stack จะ align ผิด ปัญหาทำนองเดียวกันก็เกิดบ่อยใน pointer, int รูปต่างๆ และใน floating point

## 9. Compound Statements

กลุ่มของคำสั่งที่ล้อมรอบด้วยวงเล็บปีกกา รวมเรียกว่า compound statement ตำแหน่งของการวางวงเล็บปีกกามีหลายรูปแบบ ทำให้เกิด style มากมาย หลักสำคัญคือ ยึด style ชนิดใดชนิดหนึ่งเป็นหลัก (ตามมาตรฐานที่ถนัด) ในการเขียนโปรแกรม เช่น ในการแก้ไขโปรแกรมของผู้อื่น ควรจะยึด style ที่ใช้ในโปรแกรมนั้น จะได้มีรูปแบบที่เสมอต้นเสมอปลายโดยตลอด ดังตัวอย่าง

```
control {
   statement;
   statement;
}
```

(control อาจจะแทน if, for, หรือ while ฯลฯ) Style ข้างต้นเรียกว่า “K&R Style” ซึ่งเป็นที่นิยมใช้กันมาก โดยเฉพาะสำหรับผู้ที่ยังไม่มี style ของตนเอง ในการเขียนแบบ K&R Style นั้น ส่วนอนุประโยค else ใน if-else และ while ใน do-while จะอยู่บรรทัดเดียวกับวงเล็บปีกกาเปิด แต่สำหรับ style อื่น วงเล็บปีกกาจะอยู่บนบรรทัดเดียวโดดๆ เช่น (จะแสดงให้ดูเฉพาะ 3 ตัวอย่างข้างล่างเท่านั้น ผู้เขียนนิยมอีก style)

```
if (condition) {
   statement;
   statement;
} else
   statement;
```

(อนุประโยคเป็นตัวอย่างที่ไม่ดี ดูคำอธิบายต่อไป) และ

```
do {
   statement;
   statement;
} while (condition);
```

สำหรับส่วนของโปรแกรมที่มี label หลายอัน labels เหล่านี้ควรจะอยู่บนบรรทัดเดียวโดดๆ fall-through ของ switch (คือ label ที่ไม่มี break คั่นระหว่าง labels) ต้องใส่ comment เพื่อสะดวกในการ maintenance ภายหลัง เช่น

```
switch (expression) {
case ABC:
case DEF:
   statement;
   break;
case UVW:
   statement;
   /* fall-through   */
case XYZ:
   statement;
   break;
}
```

ในที่นี้ break อันสุดท้ายไม่จำเป็น แต่ต้องมีไว้เพราะมิฉะนั้น จะเกิดกรณี fall-through เมื่อเพิ่ม label ต่อท้ายในอนาคต ถ้ามี default ให้วางไว้ท้ายสุด และไม่ต้องใส่ break

สำหรับ ***if-else*** ที่ประกอบด้วย compound statement ไม่ว่าจะเป็นอนุประโยคใน ***if*** หรือ ***else*** ควรจะใส่วงเล็บปีกกาทั้งหมด (ตัวอย่างข้างต้นเป็นแบบอย่างที่ไม่ดี เพราะส่วน else ไม่มีวงเล็บปีกกา) ลักษณะนี้เรียกว่า ***fully bracketed syntax*** เช่น

```
if (expression) {
   statement;
   statement;
} else {
   statement;
   statement;
}
```

ในกรณีของ ***if-if-else*** ที่ซ้อนๆ กัน จำเป็นต้องใส่วงเล็บปีกกาเป็นอย่างยิ่ง เพื่อป้องกันความผิดพลาด มิฉะนั้น compiler จะแปลผิด แม้ว่า โปรแกรมเมอร์จะย่อหน้าไว้อย่างที่ตนคิดว่าควรจะเป็น เช่น

```
if (expr1)
   if (expr2)
         funca();
else
   funcb();
```

ควรจะเขียนเป็น

```
if (expr1)
{
   if (expr2)
   {
         funca();
   }
}
else
{
   funcb();
}
```

ถ้าอนุประโยค else มี if ต่อเป็น ***else if*** ควรจะเขียนชิดซ้ายทั้งหมด เพื่อป้องกันการย่อหน้าเป็นขั้นบันได ยาวเกินควร

```
if (STREQ(reply, "yes"))
{
   statements for yes;
}
else if (STREQ(reply, "no"))
{
   statements for no;
}
else if (STREQ(reply, "maybe"))
{
   statements for maybe;
}
else
{
   statements for default;
}
```

รูปแบบดังกล่าวจะทำให้ดูเหมือนกับ generalized ***switch*** โดยมีย่อหน้าเป็นตัวบอกถึง case ต่างๆ แทนที่จะมองเป็น nested-if statement

ประโยค ***do-while*** ควรจะมีวงเล็บปีกกาเสมอ

ตัวอย่างต่อไปนี้ เป็นตัวอย่างที่ไม่ควรใช้เป็นอย่างยิ่ง

```
#define     CIRCUIT
#     define      CLOSE_CIRCUIT(cirno)     { close_circ(cirno); }
#else
#     define      CLOSE_CIRCUIT(cirno)
#endif
```

โดยในโปรแกรมมีการเรียกใช้ดังนี้

```
if (expr)
      statement;
else
      CLOSE_CIRCUIT(x)
++i;
```

ถ้าเมื่อใดที่ CIRCUIT ไม่ถูกนิยามไว้ ประโยคที่ตามหลัง คือ ++i จะทำงานทันที (ในส่วนของ else) ในกรณีที่ expr ไม่จริง นั่นหมายถึงโปรแกรมทำงานผิดจากความตั้งใจเดิม ตัวอย่างนี้ ยังแสดงให้เห็นถึงการใช้ macro โดยเขียนเป็นตัวพิมพ์ใหญ่ทั้งหมด และควรทำให้อยู่ในรูปของ fully-bracketed ด้วย

ในกรณีที่ ***if*** จำเป็นต้องทำให้เกิด unconditional control transfer โดย break, ***continue***, goto หรือ ***return*** และไม่มี ***else*** (เรียกว่า implicit else) ประโยคถัดมาไม่ควรจะย่อหน้า กล่าวคือ

```
if (level > limit)
   return overflow;
normal();
return level;
```

การไม่ย่อหน้าที่ประโยค normal() ซึ่งเป็น implicit else บอกให้ผู้ดูโปรแกรมทราบว่า เงื่อนไขภายใน if ไม่มีส่วนสัมพันธ์กับบรรทัดอื่นๆ หรือกล่าวอีกนัยหนึ่ง ไม่ต้องใส่ else ในกรณีที่เป็น implicit (แต่เพื่อป้องกันการเข้าใจผิด โดยเฉพาะในกรณีนี้ ควรจะใส่วงเล็บปีกกาให้ if คือรอบ return)

## 10. Operators

unary operator ไม่ควรเขียนแยกจาก operand ที่เป็นตัวเดียว โดยทั่วไป binary operator จะใช้ 1 blank คั่นระหว่าง operand ยกเว้น "." และ "->" สำหรับ expression ที่มีความสลับซับซ้อนมาก ก็ต้องใช้วิจารณญาณที่เหมาะสม เช่น operators ระดับในไม่ใส่ blank แต่ไปใส่ในระดับนอก เป็นต้น

ถ้าคิดว่า expression นั้นจะอ่านยาก ลองเขียนต่อบรรทัดไป การแยกก็ถือเอาช่วง operator ที่มี precedence ต่ำสุดเป็นเกณฑ์จะดีที่สุด เนื่องจาก C มีกฎข้อยกเว้นที่เกี่ยวกับ precedence, expression ใดที่ประกอบด้วย operators หลายๆ ชนิดผสมกัน ควรจะใส่วงเล็บ แต่ระวังอย่าใช้วงเล็บซ้อนๆ กันมาก เพราะสายตาคนไม่เก่งในเรื่องการจับคู่วงเล็บ

ในบางครั้ง comma operator ก็มีความจำเป็นเช่นกัน แต่โดยทั่วไป ถ้าเลี่ยงไม่ใช้ก็จะเป็นการดี comma operator เหมาะสมสำหรับ multiple initializations และ operations อย่างเช่นในคำสั่ง for ในกรณีของ expression ที่ซับซ้อน เช่น พวกที่มี ternary operator ?: ซ้อนอยู่ภายใน ไม่ควรจะเพิ่ม comma operator ให้เกิดความสับสนมาก

ขึ้น ตัวอย่างข้อยกเว้นกรณีนี้ได้แก่ macro getchar และ putchar ซึ่งใช้ทั้ง ternary และ comma operators, operands ของ logical expression ก่อน ?: ควรจะใส่วงเล็บให้หมด และค่าที่ส่งกลับต้องเป็น type เดียวกัน

## 11. กฎเกณฑ์การตั้งชื่อ (Naming Conventions)

ทุกโครงการจะมีข้อกำหนดในการตั้งชื่อของตนเอง โดยทั่วไปมักจะยึดแนวทางปฏิบัติคือ

- ชื่อที่ขึ้นต้นและลงท้ายด้วย underscores จะใช้สำหรับชื่อระบบเท่านั้น ไม่ควรใช้กับชื่อที่ตั้งโดย user ระบบทั่วไปจะใช้กฎเกณฑ์อันนี้ตั้งชื่อที่ user ไม่ควรจะรู้ ถ้าจำเป็นต้องตั้งชื่อที่ไม่ต้องการให้ซ้ำ ควรจะเริ่มตั้งตัวหนังสือ 1-2 ตัวที่เป็นตัวย่อของ package ที่รวบรวมชื่อเหล่านั้น
- ชื่อใน #define ควรเป็นตัวใหญ่ (uppercase) ทั้งหมด
- อักษรตัวแรกของ enum constant ควรเป็นตัวใหญ่ หรือเป็นตัวใหญ่ทั้งหมดเลยก็ได้
- ชื่อของ function, typedef, และตัวแปร เช่น struct, union, enum tag ควรใช้ตัวเล็ก (lowercase) หมด
- ชื่อ macro function ส่วนมากจะเป็นตัวใหญ่ทั้งหมด ยกเว้นบางอัน เช่น getchar และ putchar จะใช้ตัวเล็ก เพราะบางครั้งทั้งคู่ใช้ในลักษณะของ function ดังนั้นการใช้ตัวเล็กสำหรับ macro ควรใช้เฉพาะกรณีที่ macro นั้นทำหน้าที่เป็น function กล่าวคือ parameter ของ macro ถูกเรียกใช้เพียงครั้งเดียว และไม่มีการเอาค่าไปเก็บไว้ใน named parameters เป็นการยากในบางครั้งที่จะเขียน macro ซึ่งทำหน้าที่เหมือน function แม้ว่า arguments ของ macro นั้นจะถูกเรียกใช้เพียงครั้งเดียว
- หลีกเลี่ยงชื่อซ้ำกันที่ต่างเฉพาะอักษรตัวแรก เช่น *foo* และ *Foo* ทำนองเดียวกัน หลีกเลี่ยง *foobar* กับ *foo_bar* เพราะโอกาสที่จะเกิดความสับสนมีมาก
- หลีกเลี่ยงชื่อคล้ายๆ กัน เช่น ใช้ตัวอักษรที่คล้ายคลึงกัน ตัวอย่าง l, 1, I ซึ่งดูคล้ายๆ กันบนจอ โดยเฉพาะชื่อตัวแปร l (แอล) ซึ่งดูคล้ายๆ กับค่าคงที่ 1 (หนึ่ง)
- ในกรณีทั่วไป global names (รวมทั้ง enum) ควรใช้ prefix ร่วมกันเพื่อแยกว่า ตัวแปรนั้นอยู่ใน module ใดอันที่จริง global อาจจะจับรวมกันอยู่ใน global structure เดียวกัน
- ชื่อ typedef มักใช้ “_t” ต่อท้ายชื่อ
- หลีกเลี่ยงชื่อที่อาจจะซ้ำกับชื่อใน standard library โดยเฉพาะบางระบบที่มักจะ include library มากเกินความจำเป็น อาจทำให้มีชื่อตัวแปรที่ไม่ต้องการมาก ซึ่งเป็นผลเสียในการเพิ่มขยายโปรแกรมในอนาคต เนื่องจากชื่อตัวแปรใหม่ไปซ้ำกับชื่อใน library ที่เกินมา

## 12. ค่าคงที่ (Constants)

ค่าตัวเลขที่ใช้เป็นค่าคงที่ไม่ควรใช้ ควรใช้ *#define* แทน โดยตั้งชื่อที่มีความหมายสอดคล้องกับความหมายหรือจุดประสงค์ของการใช้ค่าคงที่นั้นๆ การใช้สัญลักษณ์ดังกล่าวทำให้โปรแกรมอ่านง่ายขึ้น เมื่อรวบรวมค่าเหล่านี้ไว้ที่เดียว จะทำให้การดูแลจัดการเกี่ยวกับโปรแกรมใหญ่ๆ สะดวก เพราะเพียงแต่เปลี่ยนค่าใน #define เท่านั้น ในกรณีที่ค่าคงที่เป็น set ของค่าที่ไม่ต่อเนื่อง (discrete values) ควรใช้ enum เพราะ enum มีข้อดีของการมี type checking ถ้าหลีกเลี่ยงไม่ได้จริงๆ อย่างน้อยที่สุดก็ควรมี comment อธิบายที่มาของค่าคงที่นั้น

การใช้ constant ควรนิยามให้ตรงกับชนิดของตัวแปร เช่น กำหนดค่า 540.0 สำหรับ float ไม่ควรใช้ 540 แล้วพึ่งการ implicit cast ของ compiler มีบางกรณีที่ค่าคงที่ (เช่น 0 และ 1) อาจจะใช้ในรูปของค่าคงที่โดยไม่มีการใช้ #define ช่วย เช่น index ของ *for* ในประโยค

```
for (i = f ; i < ARYBOUND; i++)
```

จะเหมาะสมกว่า

```
door_t   *front_door = open(door[i], 7);
if (front_door == 0)
         error("can't open %s\n", door[i]);
```

เพราะ front_door เป็น pointer ในการเปรียบเทียบค่า pointer ใดๆ ให้ใช้ *NULL* แทน 0 เพราะ *NULL* เป็นค่าคงที่จาก standard I/O library header (*stdio.h* หรือ *stdlib.h*) แม้แต่ในกรณีที่ใช้ 1 หรือ 0 โดดๆ ก็ควรใช้ #define เป็น *TRUE* หรือ *FALSE* (บางครั้งอาจเปลี่ยนเป็น *YES* หรือ *NO* จะเข้าใจดีกว่า)

Character constant แบบง่ายๆ ควรใช้ character literal แทนที่จะเป็นตัวเลข (หมายถึง ASCII code) character ที่เป็น non-text ไม่ควรใช้อย่างยิ่ง เพราะไม่ portable ถ้าหลีกเลี่ยงไม่ได้ โดยเฉพาะเมื่อเป็นส่วนหนึ่งของ string จะเขียนในรูปของ escape character ด้วย octal digit สามตัวแทนที่จะเหลือตัวเดียว (เช่น '\007') ถึงกระนั้นก็ดีวิธีดังกล่าวก็ยังถือว่าเป็น machine-dependent อยู่มาก

## 13. Macros

Expressions ที่ซับซ้อนอาจจะทำเป็น macro ที่เรียกใช้ parameters แต่ต้องระวังเกี่ยวกับปัญหาของ operator precedence วิธีแก้คือใส่วงเล็บรอบ parameter ทุกตัว ซึ่งเป็นวิธีเดียวที่จะป้องกันการเกิด side effects ข้อควรปฏิบัติอีกประการหนึ่งคือ ควรเขียน macro ที่ evaluate parameter เพียงครั้งเดียว เพราะ macro ส่วนมากทำงาน ต่างจาก function ทั่วๆ ไป

Macro บางกลุ่มอาจอยู่ในรูปของ function เช่น getc และ fgetc ควรใช้ macro ในการสร้าง function เพราะการเปลี่ยนแปลงใดๆ ใน macro จะส่งผลถึงตัว function โดยอัตโนมัติ แต่ต้องระวังการใช้ parameters ให้ดี ระหว่าง macro กับ function เพราะ parameters ที่ส่งให้แก่ function เป็นแบบ by-value เสมอ (C ไม่มีการส่ง parameter แบบ reference - ผู้เขียน) แต่ macro ส่งแบบ name-substitution การเขียน macro ที่ปราศจาก side effects จึงต้องคำนึงถึงการประกาศรูปแบบของ macro นั้นๆ อย่างชัดเจน ตัวแปรที่ใช้ใน macro ควรหลีกเลี่ยงตัวแปรชนิด global เพราะ global เหล่านั้น อาจถูกเลื่อนไป เนื่องจากชื่อที่อยู่ใน local บางจุด (ที่มีชื่อซ้ำกัน) macro ที่ใช้ชื่อตัวแปร หรืออ้างในด้านซ้ายมือของ assignment statement ควรจะใส่ comment กำกับไว้ ส่วน macro ที่ไม่ใช้ parameters แต่อ้างถึงตัวแปร หรือเป็น alias ของ function call ไม่ควรใส่ parameter เลย เช่น

```
#define     OFF_A()     (a_global + OFFSET)
#define     BORK()      (zork())
#define     SP3()       if(b) { int x; av = f(&x); bv += x; }
```

macro ช่วยประหยัด overhead ของ function call/return แต่ข้อเสียคือ ความยาว (ซึ่งในกรณีเช่นนี้ ควรใช้ function ดีกว่า เพราะ overhead ที่เกิดจาก function call/return แทบจะไม่มีผลอะไร) ในบางครั้ง ควรมีวิธีทำให้ compiler จบการทำงานของ macro อย่างถูกต้องโดยใส่ semicolon เพิ่ม เช่น

```
if (x == 3)
   SP3();
else
   BORK();
```

ถ้าไม่ใส่ semicolon หลัง SP3 ส่วนของ else จะกลายเป็นส่วนหนึ่งของ macro SP3 ซึ่งผิด logic ของโปรแกรมเดิม อันที่จริงแล้ว macro SP3 สามารถเขียนใหม่เป็น

```
#define  SP3() \
   do { if (b) { int x; av = f(&x); bv += x; } } while(0)
```

วิธีนี้ค่อนข้างจะอุ้ยอ้ายสำหรับ ***do-while*** ซึ่ง compiler และ tool บางตัวจะให้ warning ในการใช้ค่าคงที่ในส่วนของ while ซึ่งเป็น conditional อีกวิธีคือ การใช้ macro สำหรับ declaration เพื่อช่วยให้การเขียนโปรแกรมง่ายขึ้น เช่น

```
#ifdef      lint
      static      int         ZERO;
#else
#     define      ZERO        0
#endif
#define     STMT(stuff)       do {stuff} while(ZERO)
```

SP3 สามารถเขียนใหม่เป็น

```
#define     SP3() \
      STMT( if (b) { int x; av = f(&x); bv += x; } )
```

การใช้ STMT จะช่วยป้องกันความผิดเล็กๆ น้อยๆ เกี่ยวกับการพิมพ์ตก (เช่น semicolon) ซึ่งจะเปลี่ยนความหมายของโปรแกรมโดยสิ้นเชิง

ยกเว้น type ที่ถูก cast, sizeof และวิธีแก้ปัญหาข้างต้นแล้ว การเขียน macro ควรจะใส่เฉพาะ keywords เท่านั้น ถ้าตัว macro ล้อมรอบด้วยวงเล็บปีกกา

## 14. Conditional Compilation

การแปลภาษาในลักษณะเงื่อนไข (conditional compilation) เป็นวิธีการอย่างหนึ่งที่นิยมใช้ในกรณีของสิ่งที่เป็น machine-dependent, debugging, หรือจัดวิธีการทำงานในระหว่างการแปล ควรระวังการใช้ conditional compilation เพราะเงื่อนไขควบคุมการทำงานอาจก่อให้เกิดกรณีที่คาดไม่ถึงได้ เช่น ถ้าใช้ #ifdef ควบคุม machine dependencies ควรจะแน่ใจว่า ในกรณีที่ไม่มี machine เกี่ยวข้องผลควรจะเป็น error ไม่ใช่ยอมรับเป็น default machine หรือเช่น ถ้าใช้ #ifdef ในการทำ optimization กรณี default ควรจะเป็น code ที่ unoptimized แทนที่จะเป็นโปรแกรมที่ compile ไม่ได้ และอย่าลืมทดสอบ code ส่วนที่ไม่ได้ optimize

ข้อควรสังเกตเกี่ยวกับ #ifdef คือ compiler อาจจะอ่านส่วนดังกล่าว แม้ว่าเงื่อนไข ของ #ifdef เป็นเท็จ เพราะฉะนั้นแม้ว่า text ในส่วนที่ไม่เคยถูก compile เลย ภายใต้ #ifdef (เช่น #ifdef COMMENT) text นั้นไม่ควรจะเป็น text ใดๆ

ควรจัด #ifdef ไว้ใน header file แทนที่จะเป็น source file พยายามใช้ #ifdef ในการติดตั้งนิยาม macros ที่จะใช้บ่อยๆ ในโปรแกรม เช่น header file ที่ใช้สำหรับการเช็ค memory allocation อาจจะเขียนได้ดังนี้ (ข้ามส่วนของ realloc และ free)

```
#ifdef      DEBUG
      extern      void              *mm_malloc();
#     define      MALLOC(size)      (mm_malloc(size)
#else
      extern      void              *malloc();
#     define      MALLOC(size)      (malloc(size))
#endif
```

การใช้ conditional compilation ควรจะพยายามให้ขึ้นกับ feature ต่อ feature เท่านั้น หลีกเลี่ยงกรณี machine หรือ operating system dependencies เช่น

```
#ifdef      BSD4
      long  t = time((long *)NULL);
#endif
```

ตัวอย่างข้างต้นเป็นตัวอย่างที่ไม่ดีด้วยเหตุผลสองประการคือ ประการแรกอาจจะมีระบบ 4BSD ที่ใช้วิธีการที่กล่าวข้างต้น ประการที่สอง อาจจะมีระบบที่ไม่ใช่ BSD ซึ่งเหมาะกับวิธีการข้างต้น วิธีแก้ปัญหาคือ ใช้ *define* TIME_LONG และ *TIME_STRUCT* โดยกำหนดอันที่เหมาะสมไว้ใน configuration file เช่น *config.h*

## 15. Debugging

"*C Code, C code run. Run, code, run...PLEASE*!!!"——Barbara Tongue

ถ้าต้องการใช้ enum ควรจัดให้ enum constant ตัวแรกมีค่าไม่เป็นศูนย์ หรือเป็นค่าบอกว่าเกิด error เช่น

```
enum {STATE_ERR, STATE_START, STATE_NORMAL, STATE_END}  state_t;
enum {VAL_NEW=1, VAL_NORMAL, VAL_DYING, VAL_DEAD}  value_t;
```

ค่าที่ไม่ถูก initialized มักจะฟ้องออกมาเอง

ควรเช็คค่า return จากกรณีที่เกิด error แม้ว่าจะเป็น function ที่ไม่น่าจะ fail ก็ตาม เช่นกรณีของ close() และ fclose() ซึ่งอาจจะ fail ได้ แม้ว่า file operation ก่อนที่จะปิดทำงานได้ถูกต้องก็ตาม พยายามเขียน function เองเพื่อสามารถใช้ทดสอบความผิดพลาด และส่งค่า return เมื่อเกิดข้อผิดพลาดขึ้น หรือเลิกทำงานโดยสิ้นเชิงในลักษณะที่เป็นขั้นตอนเหมาะสม ควรมี code สำหรับ debugging และ error-checking แม้แต่ใน product สำเร็จรูป พยายามตรวจสอบ แม้แต่ error ที่เป็นไปไม่ได้ [8]

ใช้ *assert* ในการตรวจสอบว่าทุก function ได้รับค่าที่ถูกต้อง และค่าสิ้นสุดเป็นค่า well-formed

พยายามใช้ #ifdef ให้น้อยที่สุดในส่วนของ debug code เช่น ถ้า mm-alloc เป็น debugging memory allocator ให้ MALLOC เลือก allocator ที่เหมาะสมเอง ดังตัวอย่างข้างล่าง หลีกเลี่ยงการใส่ #ifdef ที่ไม่จำเป็น และพยายามแยกแยะข้อแตกต่างระหว่าง allocation call ที่กำลังถูก debug กับ memory ส่วนที่ใช้ในการ debug เท่านั้น เช่น

```
#ifdef   DEBUG
#     define      MALLOC(size)       (mm_malloc(size))
#else
#     define      MALLOC(size)       (malloc(size))
#endif
```

พยายามเช็คขอบเขต (bounds) แม้กระทั่งในกรณีที่ไม่ควรจะเกิดการ "overflow" ฟังก์ชันที่บันทึกค่าลงบน variable-sized storage ควรจะมี argument ที่กำหนด maxsize ของ destination ถ้าเกิดกรณีที่ไม่รู้ maxsize ของ destination ควรเลือกค่า "พิเศษ" สักค่าเป็น maxsize ซึ่งมีความหมายว่า ไม่มีการเช็คขอบเขต เมื่อเกิดกรณีผิดพลาดเกี่ยวกับการเช็คขอบเขตดังกล่าว ต้องแน่ใจว่า function ทำงานปกติ เช่น abort หรือ return ค่า error

โดยสรุป โปรแกรมที่เกิดข้อผิดพลาด จะล่าช้ากว่าโปรแกรมที่ถูกต้องซึ่งเป็นความจริง เช่นเดียวกับโปรแกรมที่ crash เป็นครั้งคราว หรือทำให้ข้อมูลดีเสียหาย

## 16. Portability

"*C combines the power of assembler with the portability of assembler.*"——Anonymous, alluding to Bill Thacker.

ข้อดีของ code ที่ portable นั้นเป็นที่ทราบกันดีในวงการของการเขียนโปรแกรม ในส่วนนี้จะกล่าวถึงแนวทางของการเขียน portable code คำว่า "portable" ในที่นี้หมายถึง source file ที่สามารถนำไป compile และ execute บนเครื่องต่างกัน เพียงแค่เปลี่ยน header files และ compiler flags ที่แตกต่างกันเท่านั้น Header files ส่วนมากจะประกอบด้วย #define และ typedef ซึ่งไม่เหมือนกันในทุกๆเครื่อง โดยทั่วไปแล้ว เมื่อพูดถึง "เครื่องใหม่" มักจะหมายถึงฮาร์ดแวร์คนละชนิด ระบบปฏิบัติการต่างกัน compiler ต่างกันหรือทุกอย่างที่กล่าวมาทั้งหมดต่างกันโดยสิ้นเชิง เอกสารอ้างอิง [1] กล่าวถึง style และ portability ที่ดีหลายประการ ข้อแนะนำสำหรับใช้เป็นแนวทางในการป้องกันไม่ให้เกิดความผิดพลาดเนื่องจากการ port โปรแกรมมีดังนี้

1. เขียนโปรแกรมที่ portable ก่อน โดยไม่ต้องกังวลถึงรายละเอียดของการ optimize นอกเสียจากจะหลีกเลี่ยงไม่ได้ โปรแกรมที่ optimized มักจะอ่านยาก การ optimize บนระบบของเครื่องหนึ่งอาจกลายเป็น code ที่แย่ลงสำหรับระบบของอีกเครื่องหนึ่ง พยายามบันทึกการแก้ไขทุกอย่างที่เกี่ยวข้องกับการปรับปรุง performance โดยจัดกลุ่มให้อยู่ใกล้กันมากที่สุด (เพื่อไม่ให้เกิด side effects ต่อ code ส่วนอื่น) การบันทึกควรจะอธิบายเหตุผล ความจำเป็น และหลักการทำงาน
2. ระลึกเสมอว่าบางสิ่งบางอย่างนั้น port ไม่ได้ เช่น code ที่ต้องจัดการเกี่ยวกับ register ค่าของ program status word หรือ code ที่ออกแบบเพื่อควบคุมการทำงานของฮาร์ดแวร์เฉพาะงาน เช่น I/O drive และ assembler ซึ่งในกรณีหลังมีหลายส่วนที่สามารถทำเป็น machine independent ได้

3. แยก files ของ code ที่เป็น machine-dependent ออกจาก machine independent เมื่อมีความจำเป็นต้อง port โปรแกรมไปสู่เครื่องชนิดอื่น จะทำให้แก้ไขสะดวกขึ้น อย่าลืมใส่ comment เกี่ยวกับสิ่งที่เป็น machine dependent ใน file นั้นๆ ด้วย
4. พฤติกรรมใดๆ ที่เป็น “implementation defined” ควรจะจัดให้เป็นประเภท machine (compiler) dependency โดยถือเสมอว่า compiler ทำงานเฉพาะอย่าง เพื่อให้สอดคล้องกับฮาร์ดแวร์ของเครื่องนั้นๆ
5. ระวังขนาดของ word แต่ละเครื่อง object อาจมีขนาดต่างกัน pointer ไม่จำเป็นว่าจะเท่ากับ *int* เสมอไปหรือแปลงกลับไปกลับมาได้อย่างสะดวก ตารางข้างล่างแสดงขนาดของตัวแปรชนิดพื้นฐานของ C บนฮาร์ดแวร์และ compiler ต่างๆ กัน

| type | pdp11 series | VAX/11 | 68000 family | Cray-2 | Unisys 1100 | Harris H800 | 80386 |
|---|---|---|---|---|---|---|---|
| char | 8 | 8 | 8 | 8 | 9 | 8 | 8 |
| short | 16 | 16 | 8/16 | 64(32) | 18 | 24 | 8/16 |
| int | 16 | 32 | 16/32 | 64(32) | 36 | 24 | 16/32 |
| long | 32 | 32 | 32 | 64 | 36 | 48 | 32 |
| char* | 16 | 32 | 32 | 64 | 72 | 24 | 16/32/48 |
| int* | 16 | 32 | 32 | 64(24) | 72 | 24 | 16/32/48 |
| int(*)() | 16 | 32 | 32 | 64 | 576 | 24 | 16/32/48 |

ฮาร์ดแวร์บางชนิดอาจจะมีขนาดสำหรับ type หนึ่งๆ มากกว่าขนาดเดียว ขนาดที่สามารถนำมาใช้งานขึ้นอยู่กับ compiler และ compile-time flags ตารางต่อไปนี้แสดงขนาดที่ “safe” สำหรับตัวแปรชนิดต่างๆสำหรับเครื่องส่วนใหญ่ โดยถือว่า unsigned numbers มีจำนวน bits เท่ากับ signed numbers

| Type | Minimum # Bits | No Smaller Than |
|---|---|---|
| char | 8 | |
| short | 16 | char |
| int | 16 | short |
| long | 32 | int |
| float | 24 | |
| double | 38 | float |
| any * | 14 | |
| char * | 15 | any * |
| void * | 15 | any * |

6. Pointer ชนิด void ประกันว่ามีขนาดของจำนวน bits ที่จะคงไว้ซึ่งความถูกต้องเมื่อใช้กับ pointer ที่ชี้ไปยัง object ใดๆ ทำนองเดียวกัน *void(*)()* สามารถใช้กับ pointer ที่ชี้ไปยัง function ใดๆ เช่นกัน pointerทั้ง 2 ชนิดนี้ใช้เมื่อมีความจำเป็นต้องแทนเป็น generic pointer (อาจใช้ *char** และ *char(*)()* สำหรับ compiler รุ่นเก่า) แต่อย่าลืมเปลี่ยน (cast) pointer เหล่านั้นให้กลับเป็นชนิดเดิมเมื่อต้องการจะใช้จริง
7. แม้ว่าสมมติ *int** และ *char** มีขนาดเท่ากัน ทั้งคู่มี format ต่างกัน ตัวอย่างข้างล่างแสดงให้เห็นว่า code จะให้ผลลัพธ์ผิดพลาดบนเครื่องบางชนิดทั้งๆที่ sizeof(int*) มีขนาดเท่ากับ sizeof(char*) หากใช้ผิดที่ ความผิดพลาดนั้นสืบเนื่องมาจาก free ซึ่งต้องการ *char** แต่กลับได้รับ *int** แทน

```
int   *p = (int*)malloc(sizeof(int));
free(p);
```

8. พึงระวังว่า ขนาดของ object ไม่ได้หมายถึง ขนาดความถูกต้อง (***precision***) ของ object เช่น Cray-2 อาจใช้ 64-bits เพื่อเก็บค่า ***int*** แต่ถ้าเอา ***long*** มาแปลงเป็น ***int*** โดยการ cast และเปลี่ยนกลับไปเป็น ***long*** เหมือนเดิม ผลลัพธ์ที่ได้อาจจะถูกตัดทอนลงเหลือเพียง 32 bits
9. เลขจำนวนเต็มศูนย์ (integer ***constant zero***) อาจ cast เป็น pointer ชนิดใดก็ได้ ผลลัพธ์ที่ได้คือ ***null pointer*** ของชนิดนั้นๆ และมีค่าต่างจาก pointer อื่นๆ ของชนิดเดียวกัน ด้วยเหตุนี้ null pointer จึงสามารถเปรียบเทียบเท่ากับ constant zero ได้เสมอ แต่ null pointer อาจเทียบไม่เท่ากับตัวแปรที่มีค่าเป็นศูนย์ เพราะ null pointer อาจจะไม่ถูกเก็บในรูปของศูนย์หมดทุกบิท Null pointer ต่างชนิดกัน บางครั้งมีค่าต่างกัน ในบางกรณี null pointer ของชนิดหนึ่งสามารถ cast ให้เป็น null pointer ของอีกชนิดหนึ่งได้
10. ใน ANSI compiler ถ้า pointer ชนิดเดียวกันสองตัวชี้ไปยังหน่วยความจำที่ตำแหน่งเดียวกัน pointer ทั้งคู่จะเปรียบเทียบได้เท่ากัน การ cast integer constant ที่ไม่เป็นศูนย์ให้เป็นชนิดเดียวกับ pointer ใดๆ นั้น อาจทำให้ pointer ผลลัพธ์เหมือนกับ pointer เริ่มต้น แต่สำหรับ non-ANSI compiler ถ้า pointer สองตัวชี้ไปที่หน่วยความจำเดียวกัน อาจจะเปรียบเทียบไม่เท่ากัน ตัวอย่างของ pointer ข้างล่างนี้ อาจเปรียบเทียบได้ผลลัพธ์ที่ไม่เท่ากัน หรือซ้ำร้ายอาจไม่ชี้ไปยังหน่วยความจำเดียวกัน

```
((int  *)2)
((int  *)3)
```

ถ้าต้องการ pointer “วิเศษ” นอกเหนือจาก NULL อาจทำได้โดยจัดที่เฉพาะไว้ หรือทำเสมือนว่า pointer นั้นมีสภาพเป็น machine dependence ดังตัวอย่าง

```
extern   int          x_int_dummy;    /* in x.c        */
#define  X_FIAL       (NULL)
#define  X_BUSY       (&x_int_dummy)

#define  X_FAIL       (NULL)
#define  X_BUSY       MD_PTR1         /* MD_PTR1 from “machdep.h” */
```

11. Floating point มีทั้ง ***precision*** และ ***range*** ซึ่งไม่ขึ้นกับขนาดของ object ดังนั้น เมื่อเลข floating-point ชนิด 32-bit เกิด overflow (หรือ underflow) ขึ้น มักจะเกิดที่ค่าต่างกันบนฮาร์ดแวร์ต่างชนิดกัน เช่น 4.9 คูณ 5.1 อาจจะให้ผลลัพธ์ต่างกันบนฮาร์ดแวร์ต่างชนิดกัน ซึ่งความแตกต่างจากการปัดเศษขึ้นหรือลง อาจส่งผลลัพธ์ที่ไม่คาดคิดมาก่อนได้
12. บนฮาร์ดแวร์บางชนิด ***double*** อาจจะให้จำนวน precision น้อยกว่า ***float*** ได้
13. บนฮาร์ดแวร์บางชนิด ครึ่งแรกของ ***double*** อาจจะมีค่าเป็น ***float*** ที่มีค่าเหมือนกัน แต่อย่ายึดเป็นหลักเกณฑ์มาตรฐาน
14. ระวังการใช้ signed character เพราะบนฮาร์ดแวร์ VAX บางรุ่น character จะถูก signed extended เมื่อใช้ใน expression ลักษณะดังกล่าวไม่เกิดกับฮาร์ดแวร์ชนิดอื่น โปรแกรมที่ยึดสมมุติฐานของ signed/unsigned นั้นไม่ portable เช่น array[c] อาจจะให้คำตอบผิดพลาด ถ้าค่า c ที่คิดว่าจะมีค่าเป็นบวกกลับกลายเป็นลบ (เพราะเป็น signed) ถ้าจำเป็นต้องใช้ signed หรือ unsigned character จริงๆ ให้เขียน comment ว่าเป็น SIGNED หรือ UNSIGNED ผลลัพธ์ที่ได้จาก unsigned จะถูกต้องเสมอก็ต่อเมื่อใช้ unsigned char เท่านั้น

15. หลีกเลี่ยงการตั้งสมมุติฐานเป็น ASCII ถ้าเลี่ยงไม่ได้ บันทึกเป็นกิจลักษณะและ localize ไว้ด้วยกัน ระลึกเสมอว่า character อาจจะมีขนาดใหญ่กว่า 8 bits ได้ (ดังจะเห็นได้จาก unicode)
16. โปรแกรมที่อาศัยข้อได้เปรียบของ two's complement เกี่ยวกับตัวเลขบนฮาร์ดแวร์ทั่วไป ไม่ควรจะยึดถือเป็นเกณฑ์ optimization ที่ใช้การ shift แทน arithmetic ซึ่งให้ผลเหมือนกันมักเป็นช่องโหว่ของการ port ถ้าจำเป็นต้อง port จริงๆ code ส่วนที่เป็น machine-dependent ควรจะรวมอยู่ใน #ifdef หรือส่วนของการทำงานที่ถูกควบคุมโดย #ifdef macro ควรจะชั่งน้ำหนักของความสำคัญระหว่างเวลาเล็กๆ น้อยๆที่ประหยัดโดยละเลยข้อปฏิบัติข้างต้น กับเวลาที่เสียไปในการ debug ข้อผิดพลาดที่เกิดจากสาเหตุข้างต้น เมื่อโปรแกรมถูก port จากเครื่องหนึ่งไปอีกเครื่องหนึ่ง
17. โดยทั่วไปแล้ว ถ้าขนาดของ word หรือช่วงของค่าเป็นสิ่งสำคัญของการ port ให้ใช้ typedef กำหนดนิยามใหม่ของ "ขนาด" ของ type ต่างๆ โปรแกรมขนาดใหญ่ควรจะมี header file ร่วมที่เก็บ typedef ของชนิดข้อมูลที่ใช้บ่อยๆ ซึ่งมีลักษณะ width-sensitive เพราะจะช่วยให้แก้ไขได้ง่ายขึ้น รวมทั้งการหา width-sensitive code ด้วย ข้อมูลชนิด unsigned ยกเว้น ***unsigned int*** โดยมากจะผูกติดกับ compiler เช่น ถ้า loop counter สามารถใช้ข้อมูลชนิด 16 หรือ 32 bits ให้ใช้ ***int*** เพราะ int จะเป็นข้อมูลธรรมชาติที่มีประสิทธิภาพสูงสุดสำหรับเครื่อง
18. การ align ข้อมูลก็มีความสำคัญเช่นกัน เช่น บนฮาร์ดแวร์ชนิด 4-byte integer ข้อมูลอาจเริ่มที่ address ใดๆ ที่ address คู่ หรือ address ทวีคูณของสี่ ในกรณีของ structure บางลักษณะอาจเกิดมี offset ต่างกันบนฮาร์ดแวร์ต่างชนิดกัน แม้ว่าองค์ประกอบของ structure นั้นอาจมีขนาดเท่ากันหมดบนฮาร์ดแวร์ทุกประเภท อันที่จริง structure ของ 32-bit pointer และ 8-bit character อาจมีขนาดต่างกันหมดบนเครื่องสามยี่ห้อ โดยเฉพาะสำหรับ pointer ที่ชี้ไปยัง object ใดๆ ไม่ควรจะเปลี่ยนข้ามไปมาตามอำเภอใจ การเก็บค่า integer โดยใช้ pointer ขนาด 4 bytes เริ่มต้นที่ address คี่ อาจจะทำงานได้เป็นบางครั้ง และอาจจะเกิด core dump ได้ในเช่นกัน หรือในบางกรณีอาจจะผิดแบบเงียบๆ (และไปทำลายข้อมูลส่วนอื่น) pointer สำหรับ character เป็นจุดบอดสำหรับฮาร์ดแวร์ที่ไม่สามารถ address เป็น byte ได้ การพิจารณาถึง alignment และคุณลักษณะพิเศษเฉพาะสำหรับ loader บางตัว อาจก่อให้เกิดข้อสมมุติฐานที่ผิดๆ เกี่ยวกับตัวแปรที่ประกาศติดกันว่า จะต้องถูกจัดเรียงอยู่ด้วยกันในหน่วยความจำ หรือตัวแปรชนิดหนึ่งควรจะ align ตรงกับตัวแปรชนิดอื่นโดยไม่น่าจะเกิดปัญหาแต่อย่างใด
19. แต่ละ byte ใน word จะมีความสำคัญตามลำดับเพิ่มขึ้นเรื่อยๆ ตาม address ของ byte บนเครื่องเช่น VAX ที่มีสถาปัตยกรรมแบบ little-endian หรือลดความสำคัญลงตามลำดับของ address ที่เพิ่มขึ้น เช่น ตระกูล 68000 หรือ big-endian ลำดับของ byte ใน word ที่ประกอบเป็น object ขนาดใหญ่ขึ้น (เช่น double word) อาจจะมีการจัดเรียงต่างกัน ดังนั้น โปรแกรมใดที่ผูกติดกับการจัดเรียงบิท ซ้าย-ขวา ของ object ต่างๆ จะต้องตรวจตราเป็นพิเศษ ระวัง bit fields ภายใน structure จะทำให้ portable ได้ก็ต่อเมื่อ fields เหล่านั้น ไม่ได้ถูกจัดให้ต่อกัน (concatenate) และเรียกใช้เสมือนเป็นข้อมูลก้อนเดียวกัน [1, 3] อันที่จริง การ concatenate ตัวแปรสองจำนวนเข้าด้วยกันนั้นไม่ portable
20. บางครั้งอาจจะมีช่องว่างใน structure ที่ไม่ได้ใช้งาน ซึ่ง union บางอันใช้เป็นวิธีลัดในการจัดการขนาดของข้อมูล พึงสังวรว่า ค่าใดๆ ไม่ควรจะเก็บในรูปแบบหนึ่ง (type) แล้วเรียกใช้ในอีกรูปแบบหนึ่ง การใช้ tag ใน union อาจจะเป็นตัวช่วยในการบอกว่าอะไรเป็นอะไร

21. Compiler ต่างกันมักใช้วิธีการส่งค่า structure กลับต่างกัน ปัญหาเกิดเมื่อ libraries ส่งค่าของ structure กลับมายังโปรแกรมที่แปลโดย compiler ต่างกัน อาจทำให้การจัดเรียงข้อมูลไม่ตรงกัน structure pointers กลับไม่เป็นปัญหาในการ port แต่อย่างใด
22. อย่าพยายามตั้งสมมุติฐานของการส่ง parameter เอง โดยเฉพาะขนาดของ pointer และลำดับการ evaluate, ขนาดของ parameter, ฯลฯ ตัวอย่างโปรแกรมข้างล่างเป็นตัวอย่างที่ไม่ portable

```
c = foo(getchar(), getchar());

char
foo(char c1, char c2, char c3)
{
   char  bar = *(&c1 + 1);
   return bar;                    /* often won't return c2  */
}
```

ตัวอย่างนี้มีที่ผิดมากมาย ประการแรก stack อาจจะขยายขึ้นหรือลง (ซึ่งในความเป็นจริงอาจจะไม่มี stack เลยก็ได้) ประการต่อมา parameter อาจถูกขยายให้เป็นขนาดที่ใหญ่ขึ้น (เช่น *char* เป็น *int*) ขณะที่ส่ง Arguments แต่ละตัวอาจจะถูกจัดเก็บลงบน stack โดยเรียงจากซ้ายไปขวา ขวาไปซ้าย ไม่มีลำดับก่อนหลัง หรือส่งผ่าน register (โดยไม่มีการ push เลย) ลำดับของการ evaluate อาจไม่เหมือนกับลำดับของการ push ประการสุดท้าย compiler บางตัว อาจมีวิธีการเรียก parameter ที่ไม่เหมือนชาวบ้านเลย
23. บนฮาร์ดแวร์บางประเภท null char pointer ((char *)0) มีความหมายเหมือนกับ pointer ที่ชี้ไปยัง null string อย่ายึดถือวิธีการดังกล่าวเป็นหลักปฏิบัติ
24. อย่าแก้ไขค่าของ string constant ตัวอย่างต่อไปนี้ เป็นตัวอย่างที่ไม่ดี ซึ่งมักพบกันบ่อยๆ

```
s = "/dev/tty??";
strcpy(&s[8], ttychars);
```

25. โดยทั่วไป address space อาจจะมีช่องว่างที่ไม่ได้ใช้งาน การคำนวณ address ของค่าที่ไม่ได้จัดเตรียมไว้ล่วงหน้าใน array (ก่อนและหลังการใส่ค่าลงในแต่ละช่องของ array) อาจทำให้โปรแกรมหยุดทำงานได้ ถ้า address นั้นๆ ถูกใช้ในการเปรียบเทียบ โปรแกรมอาจจะยังคงทำงานต่อได้ แต่ข้อมูลจะผิดพลาดเสียหาย ให้คำตอบผิด หรือวน loop ไม่รู้จบ ใน ANSI C pointer ที่ชี้ไปยัง array ใดอาจจะชี้ไปที่ช่องถัดจากตำแหน่งสุดท้ายของ array นั้น วิธีดังกล่าว "ปลอดภัย" สำหรับโปรแกรมรุ่นเก่า สิ่งที่พึงระวังคือ pointer ที่ชี้ไปยังช่องถัดจากท้ายของ array นั้นจะ dereference ไม่ได้เด็ดขาด
26. เฉพาะการเปรียบเทียบระหว่าง == และ != ใช้ได้กับ pointers ทุกตัวของข้อมูลชนิดใดชนิดหนึ่ง สำหรับการเปรียบเทียบประเภทอื่นๆ ที่ portable คือการเปรียบเทียบระหว่าง < , <= , > หรือ >= เมื่อ pointers ทุกตัวที่เกี่ยวข้องชี้ไปยัง array เดียวกัน (หรือช่องถัดจากท้ายของ array เท่านั้น) ทำนองเดียวกัน การทำ arithmetic operators กับ pointers จะ portable ก็ต่อเมื่อกระทำกับ pointers ที่ชี้ไปยัง array เดียวกัน (หรือช่องถัดจากท้ายของ array เท่านั้น)

27. ขนาดของ word มีผลต่อการ shift และ mask ตัวอย่างต่อไปนี้แสดงให้เห็นว่า เฉพาะ 3 bits นับจากขวาสุดของตัวแปรชนิด int บนฮาร์ดแวร์ 68000 บางรุ่น จะถูกลบเป็นศูนย์ แต่กับฮาร์ดแวร์อื่นๆ 2 bytes แรกจะถูกลบเป็นศูนย์หมด

```
X  &= 017770
```

ดังนั้นควรจะใช้

```
X  &= ~07
```

ซึ่งจะให้ผลถูกต้องบนฮาร์ดแวร์ทุกประเภท สำหรับ bit field ไม่ประสบปัญหาดังกล่าว

28. Expressions ที่ก่อให้เกิด side effects มักจะทำให้ความหมาย (semantics) ของโปรแกรมเปลี่ยนไป ขึ้นอยู่กับ compiler ที่ใช้ เนื่องจากลำดับการ evaluate ในภาษา C นั้น ส่วนมากจะไม่มีการกำหนดตายตัว (undefined) ตัวอย่างผิดๆที่เห็นกันประจำคือ

```
a[i] = b[i++];
```

เราเพียงแต่รู้ว่า subscript i ใน b ยังไม่มีการบวกเพิ่ม แต่การอ้างอิงไปสู่ array a อาจจะเป็นค่าที่ยังไม่เพิ่ม (เหมือนเดิม) หรือค่าที่เพิ่มแล้วก็ได้

```
struct      bar_t { struct bar_t *next; } bar;
bar->next = bar = tmp;
```

ในตัวอย่างที่สอง address ของ bar->next อาจจะถูกกำหนดก่อนที่ค่า tmp จะถูกนำไปเก็บใน bar

```
bar = bar->next = tmp;
```

ในตัวอย่างที่สาม bar อาจจะถูก assigned ก่อน bar->next แม้ว่าขั้นตอนดังกล่าวดูจะผิดหลักเกณฑ์ที่ว่า “กฎของ assignment จะเริ่มจากขวาไปซ้าย” แต่การตีความข้างต้นเป็นวิธีที่ชอบด้วยหลักเกณฑ์ ดังตัวอย่างประกอบ

```
long  i;
short a[N];
i = old;
i = a[i] = new;
```

ค่าที่ i ได้รับในขั้นตอนสุดท้ายจะต้องเป็น “ค่าที่ type เหมือนกับ assignment ที่เริ่มจากขวาไปซ้าย” (กล่าวคือ (long)(short)) อย่างไรก็ตาม i อาจจะได้รับค่า “(long)(short)new” ก่อน a[i] ด้วยซ้ำ ทั้งนี้ compiler ต่างกัน ก็จะกำหนดขั้นตอนการทำงานที่ต่างกัน

29. ตั้งข้อสงสัยไว้ก่อนเลยเมื่อเจอค่าตัวเลขในโปรแกรม (มักจะเป็น “magic numbers”)

30. พยายามหลีกเลี่ยงลูกเล่นที่ใช้ preprocessor ช่วย เช่นการใช้ /**/ สำหรับสร้าง token หรือ macros ที่พึ่งการแทนค่าใน argument ขณะที่ string ถูกแทนค่าผิดๆเสมอ ดังตัวอย่าง

```
#define  FOO(string) (printf("string=%s",(string)))
...
FOO(filename);
```

บางครั้งจะถูกแทนค่าเป็น

```
(printf("filename=%s",(filename)))
```

ระวังลูกเล่นที่ใช้ใน preprocessor อาจทำให้มีการแทนค่าผิดพลาดของ macros บนฮาร์ดแวร์บางประเภท ดังตัวอย่าง

```
#define  LOOKUP(chr) (a['c'+(chr)]) /*works as intended*/
#define  LOOKUP(c)   (a['c'+(c)])   /*sometimes breaks */
```

ตัวอย่างที่สองจะถูกแทนค่าได้สองลักษณะ ทำให้โปรแกรมทำงานไม่เป็นไปตามที่คาดไว้

31. ศึกษาทำความคุ้นเคยกับ library functions และ defines (แต่อย่างลงลึกมาก เพราะรายละเอียดของการทำงานที่ไม่เกี่ยวกับ interface ภายนอก อาจมีการเปลี่ยนแปลงแก้ไขโดยไม่มีการแจ้งล่วงหน้า รายละเอียดส่วนมากมักจะไม่ portable ด้วย) ไม่ควรจะเขียน routines ที่เปรียบเทียบ string, terminal control หรือกำหนด system structures เอง การทำเองเป็นการเสียเวลาโดยใช่เหตุ และทำให้โปรแกรมอ่านเข้าใจยาก เพราะคนอื่นต้องคอยพะวงว่าโปรแกรมส่วนที่เราเขียนเองมีอะไรที่แตกต่างจากสิ่งที่มีอยู่แล้ว และยังเป็นการกีดขวางการติดต่อใช้ประโยชน์จาก microcode หรือเครื่องช่วยอื่นๆของ system routines ในกรณีที่กำหนด system structure ต่างออกไปเอง นอกจากนี้สิ่งที่กำหนดขึ้นเองยังเป็นทางนำไปสู่ bug ของโปรแกรม ถ้าเป็นไปได้ ศึกษาความแตกต่างระหว่าง common libraries ของ ANSI และ POSIX
32. พยายามเรียกใช้ *lint* ถ้ามีให้ใช้เพราะเป็นเครื่องมือที่ช่วยในการหา machine-dependent รูปแบบต่างๆ รวมทั้งข้อขัดแย้งหรือ bugs ที่อาจหลุดรอดการตรวจของ compiler ถ้า compiler มี switch ที่จะเพิ่มการส่ง warning ให้ทราบใช้ให้เป็นประโยชน์มากที่สุด
33. ตั้งข้อสงสัย label ที่อยู่ภายใน block ที่อาจเกี่ยวพันกับ switch หรือ goto นอก block นั้น
34. เมื่อใดก็ตามที่เกิดความไม่แน่ใจเกี่ยวกับ type ของ parameters ให้ cast เป็น type ที่เหมาะสม ควรจะ cast NULL ทุกครั้งที่ปรากฏใน non-prototyped function calls อย่าพยายามอาศัย function calls เป็นสื่อในการเปลี่ยนชนิด (type) ของ parameters เพราะ C มีกฎการ promote ชนิดของตัวแปรที่เข้าใจยาก เช่น ถ้า function หนึ่งต้องการค่า 32-bits *long* แต่กลับได้ 16-bit *int* stack ของ function อาจจะเกิดการเรียงขนาดผิด (misaligned) ค่าที่ส่งผ่านจะเกิดการ promote ผิดไป
35. ใช้ explicit cast เมื่อมีการคำนวณที่ผสมระหว่างค่า signed และ unsigned
36. การกระโดดระหว่าง procedure เช่น *longjump* ควรจะใช้อย่างระมัดระวัง ส่วนมากมักจะลืม restore ค่าใน registers พยายามประกาศค่าที่มีความสำคัญยิ่งยวด (critical values) เป็นชนิด *volatile* ถ้าทำได้ หรืออย่างน้อยก็ใส่ comment ให้เป็น VOLATILE
37. Linkers บางตัวเปลี่ยนชื่อเป็นตัวอักษรเล็ก (lowercase) บางตัวรับแค่ 6 ตัวอักษรแรกว่าเป็น unique เพราะฉะนั้นโปรแกรมอาจจะทำงานผิดโดยที่ไม่มีอะไรกระโตกกระตากบนเครื่องเหล่านี้
38. พึงระวังสิ่งที่นอกเหนือจากกติกา (compiler extensions) ถ้าจำเป็นต้องใช้จริงๆ บันทึกไว้อย่างละเอียดและถือเป็นรูปแบบของ machine dependencies
39. โดยทั่วไป โปรแกรมจะไม่ทำงานในส่วนของ data segment หรือ บันทึกผลลัพธ์ลงใน code segment แม้ว่าบางกรณีอาจทำได้แต่ไม่มีอะไรที่จะรับประกันว่าโปรแกรมจะทำได้ถูกต้องเสมอไป

# 17. ANSI C

C compiler ในปัจจุบันสนับสนุนมาตรฐานของ ANSI C บางส่วนหรือทั้งหมด ดังนั้นควรจะเขียนโปรแกรมที่ทำงานภายใต้ standard C และใช้รูปแบบมาตรฐาน อาทิ function prototypes, constant storage และ volatile storage มาตรฐาน C เพิ่มประสิทธิภาพของโปรแกรมโดยการให้ข้อมูลที่ดีกว่าแก่ optimizers อีกทั้งยังเพิ่ม portability โดยประกันว่า compiler มาตรฐานทุกตัวจะต้องรับ input language เดียวกัน โดยอาศัยวิธีการซ่อน machine dependencies หรือแจ้งเป็น warnings สำหรับส่วนของ code ที่เข้าข่าย machine-dependent

## 17.1 Compatibility

พยายามเขียน code ที่ port ไปสู่ compiler รุ่นเก่าได้ง่าย เช่น ใช้ conditional #defined ในการกำหนด keywords ต่างๆ ใน global header files อาทิ const, volatile ฯลฯ Compiler มาตรฐานกำหนดสัญลักษณ์สำหรับ preprocessor ที่เรียกใช้ได้ตลอด คือ __STDC__ สำหรับ void* เป็นชนิดที่จัดการยากหน่อยเพราะ compiler รุ่นเก่ารู้จักแต่ void วิธีง่ายๆ คือสร้าง type ใหม่ที่เป็น machine และ compiler-dependent เช่น VOIDP ในรูปของ char* บน compiler รุ่นเก่า ดังตัวอย่าง

```
#if           __STDC__
      typedef     void        *voidp;
#     define      COMPILER_SELECTED
#endif
#ifdef            A_TARGET
#     define      const
#     define      volatile
#     define      void        int
      typedef     char        *voidp;
#     define      COMPILER_SELECTED
#endif
#ifdef            ...
       ...
#endif
#ifdef            COMPILER_SELECTED
#     undef       COMPILER_SELECTED
#else
      { NO TARGET SELECTED! }
#endif
```

สังเกตว่าใน ANSI C "#" สำหรับ preprocessor จะต้องเป็นอักษรตัวแรกของบรรทัดที่ไม่ใช่ whitespace ซึ่งต่างจาก compiler รุ่นเก่า ที่ต้องเป็นอักษรตัวแรกของบรรทัดเสมอ

ใน static function ที่มี forward declaration จะต้องใส่ storage class ให้แก่ forward declaration เท่านั้น สำหรับ compiler รุ่นเก่าต้องเป็นชนิด "*extern*" แต่ ANSI กำหนดให้เป็น "*static*" ส่วน global functions ยังคงใช้ "*extern*" เหมือนเดิม ด้วยเหตุนี้การประกาศ forward declaration ใน static functions จึงควรใช้ #define โดยมีการตั้งกรอบของ #ifdef ไว้อย่างเหมาะสม เช่น #define เป็น FWD_STATIC เป็นต้น

ทุกๆ "#ifdef NAME" จะต้องจบด้วย "#endif" หรือ "#endif /* NAME */" ไม่ใช่ "#endif NAME" การใส่ comment ต่อท้าย #endif ควรใส่เฉพาะกลุ่มที่ยาวๆ เพราะกลุ่มที่สั้นๆจะจับคู่ได้ง่ายโดยเพียงแต่ดูจากตัวโปรแกรม

Tri-graphs ใน ANSI อาจทำให้เกิด string "??" ในโปรแกรมซึ่งอาจจะผิดพลาดอย่างไม่ทราบสาเหตุ

## 17.2 Formatting

รูปแบบใน ANSI C ก็เหมือนๆกับ C ทั่วไป นอกจากข้อยกเว้นที่เด่นๆสองข้อ คือ storage qualifiers และ parameter lists

เนื่องจาก *const* และ *volatile* มี binding rules ที่แปลกกว่าชนิดอื่น object ที่เป็น const หรือ volatile ควรจะประกาศแยกแต่ละ object ไป เช่น

```
int   const *s;          /* YES      */
int   const *s, *t;      /* NO       */
```

prototype ของ functions รวมกับ parameter declaration และ parameter definition เป็นบรรทัดเดียวกัน parameter ทุกตัวควรจะบันทึกในส่วน comment ของ function

```
/*
 * 'bp': boat trying to get in.
 * 'stall': a list of stalls, never NULL
 * returns stall number, 0 => no room.
 */
int
enter_pier(boat_t const *bp, stall_t *stall)
{
      ...
}
```

## 17.3 Prototypes

ควรใช้ function prototype เพื่อให้ code มีความถูกต้องมากที่สุด อีกทั้งยังช่วยให้โปรแกรมทำงานเร็วขึ้น เป็นที่น่าสังเกตว่า การประกาศ prototype

```
extern      void  bork(char c);
```

ไม่ถูกต้องตามนิยามของ function กล่าวคือ

```
void
bork(c)
      char c;
{
      ...
}
```

เพราะ prototype กล่าวว่า c เป็นค่าที่ถูกส่งไปยัง function ในรูปธรรมชาติที่สุดของเครื่อง ซึ่งมักจะเป็นหนึ่ง byte แต่ในตัวนิยาม (แบบเก่า) กล่าวว่า c จะถูกส่งผ่านแบบ int (ตามกฎที่เรียกว่า widening) ถ้า function มี parameter ที่อาจจะถูก promote เป็นค่าชนิดอื่น ทั้งตัว caller และ callee จะต้องถูก compile ให้ผลลัพธ์ที่เหมือนกัน นั่นหมายถึง ทั้งคู่ต้องใช้ prototype หรือไม่ใช้ทั้งหมด ปัญหาข้างต้นอาจหลีกเลี่ยงโดยการออกแบบอย่างเหมาะสม เช่น bork อาจจะกำหนดใหม่ให้รับค่า int

ตัวอย่างข้างต้นอาจเขียนใหม่เป็น

```
void
bork(char c)
{
      ...
}
```

แต่การประกาศแบบใหม่จะไม่ยอมรับโดย non-ANSI compiler

วิธีง่ายๆ ที่จะเขียน external declaration ที่ใช้งานได้ทั้งในรูปแบบของ prototype และไม่ prototype ใน compilers รุ่นเก่า คือ

```
#if    __STDC__
#     define    PROTO(x)    x
#else
#     define    PROTO(x)    ()
#endif

extern          char        **ncopies  PROTO((char *s, short times));
```

จะสังเกตว่า ใน PROTO จะต้องมี () ซ้อนกันสองชั้น

ที่สุดแล้ว วิธีที่ดีที่สุดคือ เขียนโปรแกรมในรูปแบบใดรูปแบบหนึ่ง (เช่น ใช้ prototype ตลอด) เมื่อต้องการทำให้เป็นแบบ non-prototype ส่วนมากก็จะใช้ tool ในการแปลง

## 17.4 Pragmas

Pragmas ใช้สำหรับการกำหนด machine-dependent code ในส่วนที่ไม่ portable (เพราะผูกติดอยู่กับฮาร์ดแวร์) รูปแบบของ ANSI pragmas ถูกกำหนดในลักษณะที่ยากต่อการแยกออกจาก pragmas อื่นๆ เพื่อรวมไว้ใน machine-dependent headers แห่งเดียว

Pragmas มีสองกลุ่มคือ *optimization* ซึ่งอาจจะละเลยโดยไม่มีผลกระทบแต่อย่างใด กับ pragma ที่เป็นพฤติกรรมของการทำงานระบบ (required pragmas) ซึ่งไม่อาจจะละเลยได้ pragmas กลุ่มหลังควรจะคุมด้วย #ifdef เพื่อหยุดการ compile โปรแกรม ถ้า required pragma ขาดหายไป

Compiler แต่ละตัวอาจทำงานตามคำสั่งของ pragma หนึ่งๆ ในลักษณะที่ต่างกัน เช่น ตัวแรกอาจจะใช้ “haggis” ในการเริ่ม optimization ในขณะที่อีกตัวอาจจะใช้เป็นตัวบอกว่า ถ้าประโยคที่กำหนดให้ถูกเรียกใช้เมื่อใดให้หยุดการทำงานโดยสิ้นเชิง ดังนั้น ทุกครั้งที่ใช้ pragmas จะต้องคุมด้วย #ifdef ที่เป็น machine-dependent เสมอ โดยเฉพาะกับ compiler ชนิด non-ANSI อย่าลืมย่อหน้าบรรทัดที่เป็น #pragma เพราะ preprocessor รุ่นเก่าอาจจะหยุดทำงานถ้าเจอคำสั่งดังกล่าว เช่น

```
#if    defined(__STDC__) && defined(USE_HAGGIS_PRAGMA)
       #pragma(HAGGIS)
#endif
```

> *“The '#pragma' command is specified in the ANSI standard to have an arbitrary implementation-defined effect. In the GNU C preprocessor, '#pragma' first attempts to*

*run the game 'rogue'; if that fails, it tries to run the game 'hack'; if that fails, it tries to run GNU Emacs displaying the Tower of Hanoi; if that fails, it reports a fatal error. In any case, preprocessing does not continue.*"

- Manual for the GNU C preprocessor for GNU CC 1.34.

## 18. ข้อควรคำนึงพิเศษ (Special Considerations)

ในส่วนนี้ จะกล่าวถึงสิ่งที่ควร และไม่ควรปฏิบัติ

1. อย่าเปลี่ยน syntax โดยใช้ macro substitution วิธีดังกล่าวทำให้โปรแกรมอ่านยากสำหรับคนอื่น
2. อย่าใช้เลขทศนิยมในที่ซึ่งเป็นค่า discrete เช่นการใช้ ***float*** เป็น index ของ loop อันเป็นการสร้างปัญหาให้กับโปรแกรม ควรทดสอบเลขทศนิยม โดยเปรียบเทียบแบบ <= หรือ >= อย่าใช้การเปรียบเทียบแบบ == หรือ != เด็ดขาด
3. Compiler มี bugs เสมอ จุดที่มักจะเกิดปัญหาบ่อยๆ คือ structure assignment และ bitfields โดยมากจะไม่สามารถทำนายว่า compiler จะทำงานผิดพลาดในรูปแบบไหน ดังนั้นจึงควรพยายามเขียนโปรแกรม โดยหลีกเลี่ยงการใช้รูปแบบที่พบว่า เป็นจุดอ่อนของ compiler ที่ใช้ เพราะไม่มีประโยชน์ที่จะพยายามแก้ไขปัญหาด้วยตัวเอง (โดยฝืนใช้รูปแบบเหล่านั้น) แม้ว่า compiler จะได้รับการแก้ไขในภายหลัง ยกเว้นถ้าจำเป็นต้องใช้รูปแบบเหล่านั้นจริงๆ พยายามเลี่ยงการใช้งานที่พบว่าเป็น bugs ของ compiler ของรูปแบบนั้นๆ
4. อย่าหวังพึ่งเครื่องช่วยในการจัดโปรแกรม (beautifier) เพราะผู้ที่ได้รับประโยชน์สูงสุดจากการเขียนโปรแกรมที่มี style ดี คือโปรแกรมเมอร์เอง โดยเฉพาะในช่วงแรกของการออกแบบ algorithm หรือ pseudo-code เครื่องช่วยเหล่านั้น (automatic beautifiers) จะมีประโยชน์สำหรับ code ที่เสร็จสมบูรณ์ ถูกต้อง แต่จะไม่เหมาะเมื่อมีความจำเป็นที่จะต้องอาศัยการจัดการเกี่ยวกับการเว้นวรรค หรือย่อหน้าเป็นกรณีพิเศษ เพราะโปรแกรมเมอร์จะทำได้ดีกว่า ในแง่ของการวาง layout ของ function หรือ file ตามที่โปรแกรมเมอร์ตั้งใจไว้ (กล่าวคือ beautifier ไม่สามารถอ่านใจโปรแกรมเมอร์ว่าต้องการจัดช่องไฟอย่างไร) สำหรับโปรแกรมเมอร์ที่ไม่มี style แน่นอน ควรจะศึกษา style การเขียนที่ดี แทนที่จะหวังพึ่ง beautifier ตลอดเพื่อทำให้โปรแกรมอ่านง่ายขึ้น
5. การลืมเครื่องหมาย "=" ใน logical compare เป็นปัญหาสำคัญอันหนึ่ง พยายามเขียนวิธีทดสอบให้ชัดเจน โดยหลีกเลี่ยง implicit test เช่น

```
abool = bbool;
if (abool)  { ... }
```

เมื่อต้องใช้ embedded assignment ในการทดสอบ ควรเขียนแบบ explicit test เพื่อไม่ต้องกลับมาแก้ภายหลัง ลองเปรียบเทียบตัวอย่างต่อไปนี้

```
while((abool = bbool) != FALSE)  { ... }
while(abool  = bbool)            { ... }
while(abool  = bbool, abool)     { ... }
```

6. พยายาม comment ตัวแปรที่ถูกเปลี่ยนให้แตกต่างไปจากการทำงานในลักษณะปกติให้ชัดเจน หรือ code ในส่วนที่อาจจะก่อให้เกิดความผิดพลาดง่ายในระหว่าง maintenance
7. Compiler รุ่นใหม่ๆ จะจัดตัวแปรไว้ใน registers โดยอัตโนมัติ จึงควรใช้ ***register*** เท่าที่จำเป็นเพื่อเป็นเครื่องแสดงว่า ตัวแปรเหล่านั้นมีความสำคัญอย่างยิ่งยวด ในกรณีที่จำเป็นมากๆ กำหนดสัก 2-4 ค่า เป็น register และส่วนที่เหลือเป็น REGISTER ซึ่งในกลุ่มหลังนี้สามารถใช้ #define แปลงเป็น register เมื่อ port ไปบนเครื่องที่มี register เพียงพอที่จะใช้งานได้เต็มที่

## 19. Lint

***Lint*** เป็น C program checker [2, 11] ตัวหนึ่งที่ใช้ตรวจโปรแกรมภาษา C เพื่อรายงานจุดที่เกิด type ไม่ตรงกัน ความไม่สมนัยระหว่างนิยามของ function และตำแหน่งที่เรียกใช้ จุดที่อาจจะเกิด bugs ในโปรแกรม ฯลฯ การใช้ ***lint*** เป็นสิ่งที่ควรทำประจำกับทุกๆ โปรแกรม ซึ่งโครงการส่วนมากจะใช้ lint เป็นเครื่องมืออันหนึ่งในการตรวจรับงาน

การใช้ ***lint*** ให้เกิดประโยชน์สูงสุด ไม่ใช่เพียงแค่ทำเป็นเครื่องมือตรวจรับงาน แต่ควรจะถือเสมือนว่าเป็นเครื่องช่วยในการแก้ไขหรือเพิ่มเติม code ใหม่ๆ เพราะ ***lint*** สามารถหา bug ที่แฝงอยู่ในรูปการณ์ที่คาดไม่ถึง และประกันปัญหาเกี่ยวกับ portability ได้ดี Messages ต่างๆ ที่เกิดจาก ***lint*** มักเป็นตัวบอกถึงปัญหาที่อาจจะเกิดขึ้น มีเรื่องตลกเกี่ยวกับโปรแกรมที่ลืมใส่ argument ในคำสั่ง fprintf ดังนี้

```
fprintf("Usage: foo  -bar <file>\n");
```

ตัวผู้สร้างโปรแกรมไม่เคยประสพปัญหา แต่สำหรับผู้ใช้คนอื่นจะเจอ core dump ทุกครั้งที่พิมพ์คำสั่งจาก command line ผิด ซึ่ง ***lint*** จะสามารถแจ้งข้อผิดพลาดข้างต้นให้ทราบทันที

Options ทั้งหลายของ ***lint*** นั้นมีประโยชน์ทั้งสิ้น บาง options อาจจะเตือนจุกจิกแม้กระทั่งสิ่งที่ถูกต้อง แต่นั่นแหละเป็นส่วนที่ทำให้เจอข้อผิดพลาดที่ไม่คาดฝัน เช่น -p จะตรวจว่า การเรียก function มี type-consistency ต่างกันอย่างไรสำหรับ library routines บางกลุ่ม ช่วยให้ program มี “coverage” ของการตรวจเช็คดีที่สุด

***Lint*** ยังสามารถตรวจ comment พิเศษๆ ในส่วนของโปรแกรมที่ทำให้ lint เตือนจุกเตือนจิก comment พิเศษอาจทำให้เลิกเตือนได้ Comment ลักษณะนี้ยังช่วยอธิบาย code พิเศษเหล่านี้ด้วย

## 20. Make

เครื่องมือที่มีประโยชน์อีกชิ้นหนึ่ง คือ make [7] ในระหว่างการพัฒนาโปรแกรม ***make*** จะช่วย recompile ส่วนของโปรแกรมที่ได้รับการแก้ไขใหม่ (จากการ ***make*** ครั้งล่าสุด) เท่านั้น Make ยังสามารถใช้ช่วยในการทำงานอื่นๆ โดยอัตโนมัติ อาทิ

| | |
|---|---|
| all | ทำให้เป็น binary ทั้งหมดเสมอ |
| clean | ลบ intermediate file ทิ้ง |
| debug | สร้าง binary “a.out” หรือ “debug” เพื่อใช้ทดสอบ |
| depend | สร้าง transitive dependencies |

| | |
|---|---|
| install | ติดตั้ง binaries และ libraries ฯลฯ |
| deinstall | เพิกถอนการติดตั้ง |
| mkcat | ติดตั้ง manual pages |
| lint | สั่งให้ Lint ทำการตรวจโปรแกรม |
| print/list | พิมพ์ source ของโปรแกรมออกกระดาษ |
| shar | รวม source โปรแกรมทั้งหมดเป็น file เดียว |
| spotless | สั่งทำการ clean แล้วเรียก revision control เพื่อจัดเก็บ sources |
| | ข้อสังเกต: จะไม่ลบ Makefile แม้ว่าจะเป็นส่วนหนึ่งของ source ก็ตาม |
| source | undo สิ่งที่ spotless ได้ทำไปทั้งหมด |
| tags | สั่ง ctags ทำงาน (อาจใช้ -t ด้วยก็ได้) |
| rdist | ส่ง source ไปยังเครื่องอื่นๆ |
| file.c | ทำการ checkout file ที่ควบคุมโดยระบบ revision control |

นอกจากนี้ การใส่ define จาก command-line สามารถกำหนดค่าต่างๆ ที่เกี่ยวกับ Makefile (เช่น “CFLAGS”) หรือค่า define อื่นๆ ในโปรแกรม (เช่น “DEBUG”)

## 21. มาตรฐานเฉพาะงาน (Project-Dependent Standards)

แต่ละโครงการอาจจะมีการกำหนดมาตรฐานนอกเหนือจากที่กล่าวมาทั้งหมด หัวข้อต่อไปนี้เป็นสิ่งที่ต้องพิจารณาโดยกลุ่มผู้บริหารโครงการ

1. ควรใช้ระบบการตั้งชื่อตัวแปรอื่นหรือไม่ โดยเฉพาะการจัดระบบ prefix เพื่อรวบรวม global data ตาม functional grouping รวมทั้ง member ของ structure หรือ union ด้วย
2. การจัดการเกี่ยวกับ include file ที่เหมาะสมกับขั้นตอนโครงสร้างของข้อมูลโครงการ
3. ควรมีขั้นตอนอย่างไรในการพิจารณาแก้ไขสิ่งผิดพลาด หรือเตือนจาก lint ถ้าจัดตั้งเกณฑ์อะลุ่มอล่วยในการแก้ไข ควรจะให้สัมพันธ์กับ option ต่างๆ ของ lint แทนที่จะปล่อยให้การเตือนที่ไม่สำคัญผ่านไป ซึ่งบางครั้งอาจรวมไปถึงคำเตือนที่สำคัญๆ เกี่ยวกับ bug หรือข้อขัดแย้งด้วยก็เป็นได้
4. ถ้ามีการจัดเก็บ library ของโครงการเอง ควรจะวางแผนในการจัดหา lint library file ให้แก่ผู้จัดการระบบ Lint library file ดังกล่าว เป็นเครื่องช่วยตรวจความสัมพันธ์ของการใช้ library functions ต่างๆ ว่าเข้ากันได้ดีหรือไม่
5. ควรใช้ระบบ revision control แบบใดจึงจะเหมาะสมกับโครงการ

## 22. บทสรุป (Conclusion)

จากมาตรฐานเกี่ยวกับรูปแบบของโปรแกรมภาษาซีที่กล่าวมาทั้งหมด จุดสำคัญๆ หลักๆ คือ

1. การเว้นช่องไฟที่เหมาะสม และ comment โดยที่ทำให้มองเห็นโครงสร้างของโปรแกรมจาก layout ตัวโปรแกรม การใช้ expression แบบง่ายๆ คำสั่ง และ functions ต่างๆ เพื่อให้เข้าใจง่ายที่สุด
2. ระลึกเสมอว่า ใครสักคน หรืออาจจะเป็นตัวเราเอง ต้องถูกเรียกให้กลับมานั่งดัดแปลงโปรแกรมหรือทำให้ใช้งานได้บนฮาร์ดแวร์ประเภทอื่นในอนาคต พยายามจัดสร้างโปรแกรมที่ port ง่าย รวมกลุ่ม (localize)

optimization สะดวก เพราะสิ่งเหล่านี้มักจะเป็นหอกข้างแคร่ (pessimizations) ในการ port ของเครื่องหรือฮาร์ดแวร์ต่างชนิดกัน

3. การเลือกใช้รูปแบบนั้นขึ้นอยู่กับผู้ใช้งาน ควรมีรูปแบบที่ consistent ระหว่างทีมงานในกลุ่ม ดีกว่าคอยเดินตามกฎเกณฑ์ตายตัว การผสมผสานรูปแบบหลายๆ ชนิดเข้าด้วยกัน เป็นสิ่งที่แย่ยิ่งกว่าการใช้รูปแบบที่ไม่ดีเพียงชนิดเดียว เพราะอย่างน้อยก็สับสนน้อยกว่า

เช่นเดียวกับมาตรฐานอื่นๆ ทั่วไป มาตรฐานข้างต้นจะเกิดประโยชน์ก็ต่อเมื่อผู้ใช้ปฏิบัติตาม ถ้ารู้สึกขัดเขินในการพยายามใช้มาตรฐานข้างต้น ปรึกษาผู้รู้ใกล้ตัว ภายในองค์กร หรือสถาบันว่าจะปรับเปลี่ยนให้เข้ากับสภาพแวดล้อมให้เหมาะสมได้อย่างไร

# บรรณานุกรม (References)


[1] B.A. Tague, *C Language Portability*, Sept 22, 1977. This document issued by department 8234 contains three memos by R.C. Haight, A.L. Glasser, and T.L. Lyon dealing with style and portability.

[2] S.C. Johnson, *Lint, a C Program Checker*, USENIX UNIX Supplementary Documents, November 1986.

[3] R.W. Mitze, *The 3B/PDP-11 Swabbing Problem*, Memorandum for File, 1273-770907.01MF, September 14, 1977.

[4] R.A. Elliott and D.C. Pfeffer, *3B Processor Common Diagnostic Standards- Version 1*, Memorandum for File, 5514-780330.01MF, March 30, 1978.

[5] R.W. Mitze, *An Overview of C Compilation of UNIX User Processes on the 3B*, Memorandum for File, 5521-780329.02MF, March 29, 1978.

[6] B.W. Kernighan and D.M. Ritchie, *The C Programming Language*, Prentice Hall 1978, Second Ed. 1988, ISBN 0-13-110362-8.

[7] S.I. Feldman, Make - *A Program for Maintaining Computer Programs*, USENIX UNIX Supplementary Documents, November 1986.

[8] Ian Darwin and Geoff Collyer, *Can't Happen or /* NOTREACHED */ or Real Programs Dump Core*, USENIX Association Winter Conference, Dallas 1985 Proceedings.

[9] Brian W. Kernighan and P. J. Plauger, *The Elements of Programming Style*, McGraw-Hill, 1974, Second Ed. 1978, ISBN 0-07-034-207-5.

[10] J. E. Lapin, *Portable C and UNIX System Programming*, Prentice Hall 1987, ISBN 0-13-686494-5.

[11] Ian F. Darwin, *Checking C Programs with lint*, O'Reilly & Associates, 1989. ISBN 0-937175-30-7.

[12] Andrew R. Koenig, *C Traps and Pitfalls*, Addison-Wesley, 1989. ISBN 0-201-17928-8.

# The Ten Commandments for C Programmers (Annotated Edition)

Henry Spencer
University of Toronto Zoology
henry@zoo.toronto.edu

1. **Thou shalt run lint frequently and study its pronouncements with care, for verily its perception and judgement oft exceed thine.**

   This is still wise counsel, although many modern compilers search out many of the same sins, and there are often problems with lint being aged and infirm, or unavailable in strange lands. There are other tools, such as Saber C, useful to similar ends.

   ``Frequently'' means thou shouldst draw thy daily guidance from it, rather than hoping thy code will achieve lint 's blessing by a sudden act of repentance at the last minute. De-linting a program which has never been lint ed before is often a cleaning of the stables such as thou wouldst not wish on thy worst enemies. Some observe, also, that careful heed to the words of lint can be quite helpful in debugging.

   ``Study'' doth not mean mindless zeal to eradicate every byte of lint output if for no other reason, because thou just canst not shut it up about some things but that thou should know the cause of its unhappiness and understand what worrisome sign it tries to speak of.

2. **Thou shalt not follow the NULL pointer, for chaos and madness await thee at its end.**

   Clearly the holy scriptures were mis-transcribed here, as the words should have been ``null pointer'', to minimize confusion between the concept of null pointers and the macro NULL (of which more anon). Otherwise, the meaning is plain. A null pointer points to regions filled with dragons,

demons, core dumps, and numberless other foul creatures, all of which delight in frolicing in thy program if thou disturb their sleep. A null pointer doth not point to a 0 of any type, despite some blasphemous old code which impiously assumes this.

3. **Thou shalt cast all function arguments to the expected type if they are not of that type already, even when thou art convinced that this is unnecessary, lest they take cruel vengeance upon thee when thou least expect it.**

A programmer should understand the type structure of his language, lest great misfortune befall him.

Contrary to the heresies espoused by some of the dwellers on the Western Shore, `int' and `long' are not the same type. The moment of their equivalence in size and representation is short, and the agony that awaits believers in their interchangeability shall last forever and ever once 64-bit machines become common.

Also, contrary to the beliefs common among the more backward inhabitants of the Polluted Eastern Marshes, `NULL' does not have a pointer type, and must be cast to the correct type whenever it is used as a function argument.

(The words of the prophet ANSI, which permit NULL to be defined as having the type `void *', are oft taken out of context and misunderstood. The prophet was granting a special dispensation for use in cases of great hardship in wild lands. Verily, a righteous program must make its own way through the Thicket Of Types without lazily relying on this rarely-available dispensation to solve all its problems. In any event, the great deity Dmr who created C hath wisely endowed it with many types of pointers, not just one, and thus it would still be necessary to convert the prophet's NULL to the desired type.)

It may be thought that the radical new blessing of ``prototypes'' might eliminate the need for caution about argument types. Not so, brethren.

Firstly, when confronted with the twisted strangeness of variable numbers of arguments, the problem returns... and he who has not kept his faith strong by repeated practice shall surely fall to this subtle trap. Secondly, the wise men have observed that reliance on prototypes doth open many doors to strange errors, and some indeed had hoped that prototypes would be decreed for purposes of error checking but would not cause implicit conversions. Lastly, reliance on prototypes causeth great difficulty in the Real World today, when many cling to the old ways and the old compilers out of desire or necessity, and no man knoweth what machine his code may be asked to run on tomorrow.

4. **If thy header files fail to declare the return types of thy library functions, thou shalt declare them thyself with the most meticulous care, lest grievous harm befall thy program.**

The prophet ANSI, in her wisdom, hath added that thou shouldst also scourge thy Suppliers, and demand on pain of excommunication that they produce header files that declare their library functions. For truly, only they know the precise form of the incantation appropriate to invoking their magic in the optimal way.

The prophet hath also commented that it is unwise, and leads one into the pits of damnation and subtle bugs, to attempt to declare such functions thyself when thy header files do the job right.

5. **Thou shalt check the array bounds of all strings (indeed, all arrays), for surely where thou typest ``foo'' someone someday shall type ``supercalifragilisticexpialidocious''.**

As demonstrated by the deeds of the Great Worm, a consequence of this commandment is that robust production software should never make use of gets() , for it is truly a tool of the Devil. Thy interfaces should always inform thy servants of the bounds of thy arrays, and servants who spurn such

advice or quietly fail to follow it should be dispatched forthwith to the Land Of Rm, where they can do no further harm to thee.

6. **If a function be advertised to return an error code in the event of difficulties, thou shalt check for that code, yea, even though the checks triple the size of thy code and produce aches in thy typing fingers, for if thou thinkest ``it cannot happen to me'', the gods shall surely punish thee for thy arrogance.**

   All true believers doth wish for a better error-handling mechanism, for explicit checks of return codes are tiresome in the extreme and the temptation to omit them is great. But until the far-off day of deliverance cometh, one must walk the long and winding road with patience and care, for thy Vendor, thy Machine, and thy Software delight in surprises and think nothing of producing subtly meaningless results on the day before thy Thesis Oral or thy Big Pitch To The Client.

   Occasionally, as with the ferror() feature of stdio , it is possible to defer error checking until the end when a cumulative result can be tested, and this often produceth code which is shorter and clearer. Also, even the most zealous believer should exercise some judgement when dealing with functions whose failure is totally uninteresting... but beware, for the cast to void is a two-edged sword that sheddeth thine own blood without remorse.

7. **Thou shalt study thy libraries and strive not to re-invent them without cause, that thy code may be short and readable and thy days pleasant and productive.**

   Numberless are the unwashed heathen who scorn their libraries on various silly and spurious grounds, such as blind worship of the Little Tin God (also known as ``Efficiency''). While it is true that some features of the C libraries were ill-advised, by and large it is better and cheaper to use the works of others than to persist in re-inventing the square wheel. But thou should take the greatest of care to understand what thy libraries promise, and what they

do not, lest thou rely on facilities that may vanish from under thy feet in future.

8. **Thou shalt make thy program's purpose and structure clear to thy fellow man by using the One True Brace Style, even if thou likest it not, for thy creativity is better used in solving problems than in creating beautiful new impediments to understanding.**

These words, alas, have caused some uncertainty among the novices and the converts, who knoweth not the ancient wisdoms. The One True Brace Style referred to is that demonstrated in the writings of the First Prophets, Kernighan and Ritchie. Often and again it is criticized by the ignorant as hard to use, when in truth it is merely somewhat difficult to learn, and thereafter is wonderfully clear and obvious, if perhaps a bit sensitive to mistakes.

While thou might think that thine own ideas of brace style lead to clearer programs, thy successors will not thank thee for it, but rather shall revile thy works and curse thy name, and word of this might get to thy next employer. Many customs in this life persist because they ease friction and promote productivity as a result of universal agreement, and whether they are precisely the optimal choices is much less important. So it is with brace style.

As a lamentable side issue, there has been some unrest from the fanatics of the Pronoun Gestapo over the use of the word ``man'' in this Commandment, for they believe that great efforts and loud shouting devoted to the ritual purification of the language will somehow redound to the benefit of the downtrodden (whose real and grievous woes tendeth to get lost amidst all that thunder and fury). When preaching the gospel to the narrow of mind and short of temper, the word ``creature'' may be substituted as a suitable pseudoBiblical term free of the taint of Political Incorrectness.

9. Thy external identifiers shall be unique in the first six characters, though this harsh discipline be irksome and the years of its necessity stretch before thee seemingly without end, lest thou tear thy hair out and go mad on that fateful day when thou desirest to make thy program run on an old system.

> Though some hasty zealots cry ``not so; the Millenium is come, and this saying is obsolete and no longer need be supported'', verily there be many, many ancient systems in the world, and it is the decree of the dreaded god Murphy that thy next employment just might be on one. While thou sleepest, he plotteth against thee. Awake and take care.
>
> It is, note carefully, not necessary that thy identifiers be limited to a length of six characters. The only requirement that the holy words place upon thee is uniqueness within the first six. This often is not so hard as the belittlers claimeth.

10. Thou shalt foreswear, renounce, and abjure the vile heresy which claimeth that ``All the world's a VAX'', and have no commerce with the benighted heathens who cling to this barbarous belief, that the days of thy program may be long even though the days of thy current machine be short.

> This particular heresy bids fair to be replaced by ``All the world's a Sun'' or ``All the world's a 386'' (this latter being a particularly revolting invention of Satan), but the words apply to all such without limitation. Beware, in particular, of the subtle and terrible ``All the world's a 32-bit machine'', which is almost true today but shall cease to be so before thy resume grows too much longer.

# ประวัติของผู้เรียบเรียง

สำเร็จ วศ.บ. (วิศวกรรมอุตสาหการ) จากจุฬาลงกรณ์มหาวิทยาลัย เป็นวิศวกรประจำโรงงาน บริษัท Parke-Davis Warner-Lambert Adams จากนั้นไปศึกษาต่อที่สหรัฐฯ สำเร็จปริญญาโท สาขาวิศวกรรมอุตสาหการ และปริญญาโท สาขาวิทยาศาสตร์คอมพิวเตอร์ จาก University of Texas at Arlington เคยเป็นผู้บริหารระบบของบริษัท Digital Matirx Systems, Inc., U.S.A. และสำเร็จปริญญาเอก สาขาวิทยาศาสตร์คอมพิวเตอร์ จาก Arizona State University

ปัจจุบัน ดร.พีระพนธ์ โสพัศสถิตย์ เป็นอาจารย์ประจำภาควิชาคณิตศาสตร์ คณะวิทยาศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย