

你所不知道的 C 語言：前置處理器應用篇

邁向專業程式設計必備技能

Copyright (憲C) 2016, 2022 宅色夫

[直播錄影](#)

概況

- 回顧 C99/C11 的前置處理器 (preprocessor) 特徵，探討 C11 新的關鍵字 `_Generic` 搭配巨集來達到 C++ template 的作用
- 探討 C 語言程式的物件導向程式設計、抽象資料型態 (ADT) / 泛型程式設計 (Generics)、程式碼產生器、模仿其他程式語言，以及用前置處理器搭配多種工具程式的技巧
- Linux 核心原始程式碼善用巨集來擴充
 - `BUILD_BUG_ON_ZERO` 巨集
 - `max`, `min` 巨集
 - `container_of` 巨集
 - Linux 核心模組 (掛載機制)

不要小看 preprocessor

怎樣讓你的程式師朋友懷疑人生：把他代碼裡的分號換成希臘文的問號：

- ; Greek Question Mark
- ; Semicolon

把以下加進 Makefile 的 CFLAGS 中:

```
' -D' ; '=' ; '
```

會看到以下悲劇訊息:

```
<command-line>:0:1: error: macro names must be identifiers
```

C++, 叫我如何接納你？

```
using net_type = loss_multiclass_log<
    fc<10,
    relu<fc<84,
    relu<fc<120,
    max_pool<2,2,2,2,relu<con<16,5,5,1,1,
    max_pool<2,2,2,2,relu<con<6,5,5,1,1,
    input<matrix<unsigned char>>
    >>>>>>>>>;
```

source: [dlib](#)

純 C 還是最美！

前置處理器是後來才納入 C 語言的特徵

由 Dennis M. Ritchie (以下簡稱 dmr) 開發的早期 C 語言編譯器 沒有明確要求 function prototype 的順序。dmr 在 1972 年發展的早期 C 編譯器，原始程式碼後來被整理在名為“last1120c”磁帶中，我們如果仔細看 [c00.c 這檔案](#)，可發現位於第 269 行的 `mapch(c)` 函式定義，在沒有 `forward declaration` 的狀況下，就分別於第 246 行和第 261 行呼叫，奇怪吧？

而且只要瀏覽 last1120c 裡頭其他 C 語言程式後，就會發現根本沒有 `#include` 或 `#define` 這一類 C preprocessor 所支援的語法，那到底怎麼編譯呢？在回答這問題前，摘錄 Wikipedia 頁面的訊息：

As the C preprocessor can be invoked separately from the compiler with which it is supplied, it can be used separately, on different languages.

Notable examples include its use in the now-deprecated `imake` system and for preprocessing Fortran.

原來 C preprocessor 以獨立程式的形式存在，所以當我們用 gcc 或 cl (Microsoft 開發工具裡頭的 C 編譯器) 編譯給定的 C 程式時，會呼叫 cpp (伴隨在 gcc 專案的 C preprocessor) 一類的程式，先行展開巨集 (macro) 或施加條件編譯等操作，再來才會出動真正的 C 語言編譯器 (在 gcc 中叫做 cc1)。值得注意的是，1972-1973 年間被稱為“Very early C compilers”的實作中，不存在 C preprocessor (!)，當時 dmr 等人簡稱 C compiler 為“cc”，此慣例被沿用至今，而無論原始程式碼有幾個檔案，在編譯前，先用 cat 一類的工具，將檔案串接為單一檔案，再來執行“cc”以便輸出對應的組合語言，之後就可透過 assembler (組譯器，在 UNIX 稱為“as”) 轉換為目標碼，搭配 linker (在 UNIX 稱為“ld”) 則輸出執行檔。

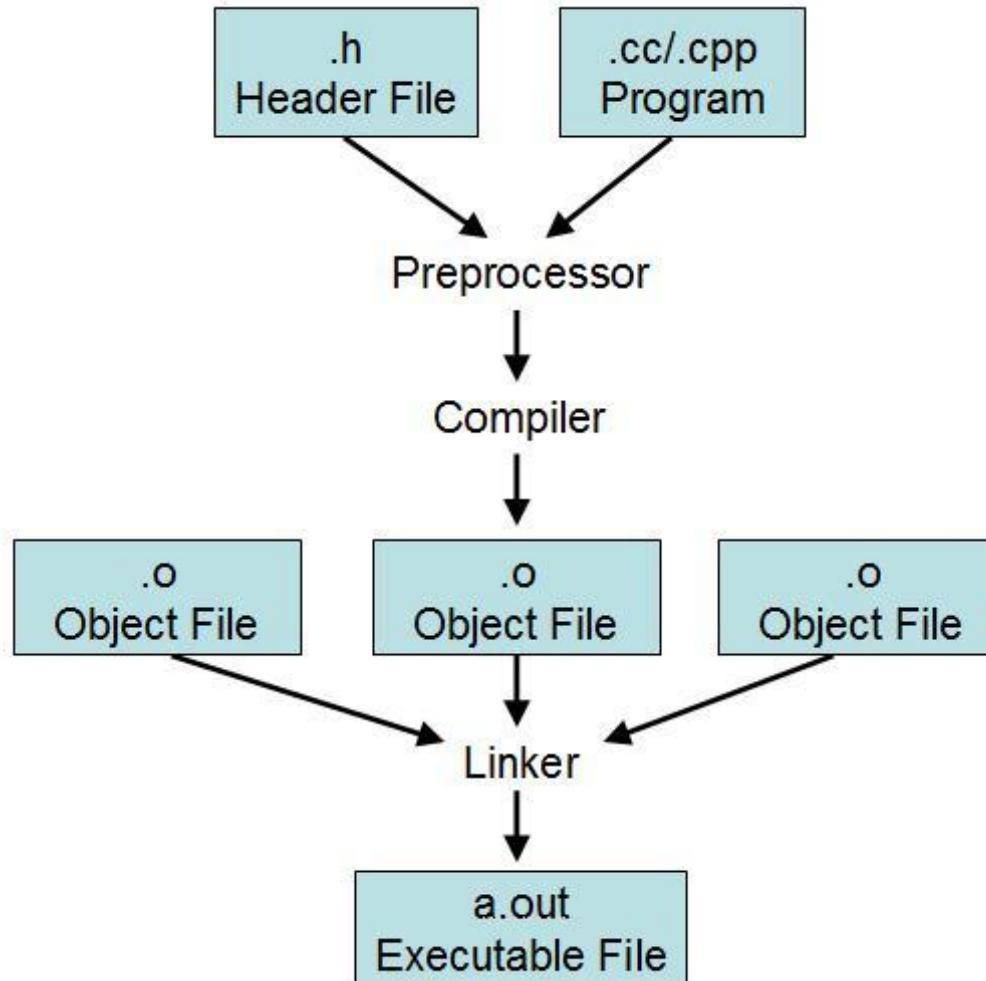
所以說，在早期的 C 語言編譯器，強制規範 function prototype 及函式宣告的順序是完全沒有必要的，要到 1974 年 C preprocessor 才正式出現在世人面前，儘管當時的實作仍相當陽春，可參見 dmr 撰寫的〈[The Development of the C Language](#)〉。C 語言的標準化是另一段漫長的旅程，來自 Bell Labs 的火種，透過 UNIX 來到研究機構和公司行號，持續影響著你我所處的資訊社會。

function prototype 的規範有什麼好處呢？可參見 [你所不知道的 C 語言：函式呼叫篇](#)。

開發物件導向程式時，善用前置處理器可大幅簡化開發

延續「[物件導向程式設計篇](#)」的思考，我們可善用前置處理器，讓程式碼更精簡、更容易維護，從而提昇程式設計思考的層面

- `#` : [Stringification/Stringizing \(字串化\)](#): 讓一個表示式變成字串，在 `assert` 巨集用到
- `# #` : [concatenation \(連結，接續\)](#)



延伸閱讀: [你所不知道的 C 語言 : 編譯器和最佳化原理篇](#)

以 raytracing (光影追蹤) 程式為例 [[source](#)], 考慮以下 macro ([objects.h](#)):

```

#define DECLARE_OBJECT(name) \
    struct __##name##_node; \

```

```

typedef struct __##name##_node *name##_node; \
struct __##name##_node { \
    name element; \
    name##_node next; \
}; \
void append_##name(const name *X, name##_node *list); \
void delete_##name##_list(name##_node *list);

DECLARE_OBJECT(light)
DECLARE_OBJECT(rectangular)
DECLARE_OBJECT(sphere)

```

light 在 `DECLARE_OBJECT(light)` 中會取代 name，因此會產生以下程式碼：

```

struct __light_node;
typedef struct __light_node *light_node;
struct __light_node { light element; light_node next; };
void append_light(const light *X, light_node *list);
void delete_light_list(light_node *list);

```

巨集實際的作用: generate (產生/生成) 程式碼。

可用 `gcc -E -P` 觀察輸出:

`_POSIX_SOURCE`

- 因為 Macro 定義的不同，會導致程式行為的不同。
- Feature Test Macros

The exact set of features available when you compile a source file is controlled by which feature test macros you define.

```
typedef enum { NORTH, SOUTH, EAST, WEST} Direction;
typedef struct {
    char *description;
    int (*init) (void *self);
    void (*describe) (void *self);
    void (*destroy) (void *self);
    void *(*move) (void *self, Direction direction);
    int (*attack) (void *self, int damage);
} Object;
int Object_init(void *self);
void Object_destroy(void *self);
void Object_describe(void *self);
void *Object_move(void *self, Direction direction);
int Object_attack(void *self, int damage);
void *Object_new(size_t size, Object proto, char *description);
#define NEW(T, N) Object_new(sizeof(T), T##Proto, N)
#define _N proto.N
```

_Generic [C11]

C11 沒有 C++ template，但有以巨集為基礎的 type-generic functions，主要透過 `_Generic` 關鍵字。

C11 standard (ISO/IEC 9899:2011): 6.5.1.1 Generic selection (page: 78-79)

以下求開立方根 (`cube root`) 的程式 (`generic.c`) 展示 `_Generic` 的使用:

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

```

#define cbrt(X) \
    _Generic((X), \
              long double: cbrtl, \
              default: cbrt, \
              const float: cbrtf, \
              float: cbrtf \
            ) (X)

int main(void)
{
    double x = 8.0;
    const float y = 3.375;
    printf("cbrt(8.0) = %f\n", cbrt(x));
    printf("cbrtf(3.375) = %f\n", cbrt(y));
}

```

編譯並執行:

```

$ gcc -std=c11 -o generic generic.c -lm
$ ./generic
cbrt(8.0) = 2.000000
cbrtf(3.375) = 1.500000

```

`<math.h>` 的原型宣告:

```

double cbrt(double x);
float cbrtf(float x);

```

- `x` (型態為 `double`) => `cbrt(x)` => selects the default `cbrt`
- `y` (型態為 `const float`) => `cbrt(y)` =>

- gcc: converts const float to float, then selects cbrtf
- clang: selects cbrtf for const float

注意 casting 的成本和正確性議題

C11 程式碼:

```
#include <stdio.h>
void funci(int x) { printf("func value = %d\n", x); }
void funcc(char c) { printf("func char = %c\n", c); }
void funcdef(double v) { printf("Def func's value = %lf\n", v); }
#define func(X) \
    _Generic((X), \
              int: funci, char: funcc, default: funcdef \
            ) (X)
int main()
{
    func(1);
    func('a');
    func(1.3);
    return 0;
}
```

輸出結果是

```
func value = 1
func value = 97
Def func's value = 1.300000
```

相似的 C++ 程式碼:

```
template <typename T> void func(T v) { cout << "Def func's value = " << v << endl;
template <> void func<int>(int x) { printf("func value = %d\n", x); }
template <> void func<char>(char c) { printf("func char = %c\n", c); }
```

對照輸出，對於 character 比對不一致，會轉型為 int。

```
func value = 1
func char = a
Def func's value = 1.3
```

[<tgmath.h>](#) - type-generic macros

[__builtin_tgmath](#) (functions, arguments)

C99 standard, 7.22 Type-generic math [<tgmath.h>](#)

libm 的 [<tgmath.h>](#) 實做

延伸閱讀:

- [C11 - Generic Selections](#)
- [Fun with C11 generic selection expression](#)
- [Experimenting with _Generic\(\) for parametric constness in C11](#)

Block

- 巨集限制很多，因為本質是「展開」，這會導致多次的 evalution
- 考慮以下程式碼:

```
#define DOUBLE(a) ((a) + (a))
int foo() {
    printf(__func__);
    return 3;
}

int main () {
    return DOUBLE(foo()); /* 呼叫 2 次 foo() */
}
```

為此，我們可以使用區域變數，搭配 GNU extension `__typeof__`，改寫上述巨集：

```
#define DOUBLE(a) ({ \
    __typeof__(a) _x_in_DOUBLE = (a); \
    _x_in_DOUBLE + _x_in_DOUBLE; \
})
```

為什麼有 `_x_in_DOUBLE` 這麼不直覺的命名呢？因為如果 `a` 的表示式中恰好存在與上述的區域變數同名的變數，那麼就會發生悲劇。

如果你的編譯器支援 **Block**，比方說 clang，就可改寫為：

```
#define DOUBLE(a) \
(^(__typeof__(a) x){ return x + x; })(a)
```

- Block in C uses a lambda expression-like syntax to create closures.
- 在 clang 使用時，要加上 `-fblocks` 編譯選項

不夠謹慎的 `ARRAY_SIZE` 巨集

- 考慮以下的使用案例:

```
void foo(int (*a)[5]) {
    int nb = ARRAY_SIZE(*a);
}
```

- Linux Kernel: `ARRAY_SIZE()`
- 藝術與核心
- 從 Linux 核心「提煉」出的 `array_size`
- `_countof` Macro

`do { ... } while(0)` 巨集

- 避開 dangling else
- 延伸閱讀: multi-line macro: `do/while(0)` vs scope block

應用: String switch in C

```
#define STRING_SWITCH_L(s) \
    switch ((*((int32_t *) (s)) | 0x20202020))
#define MULTICHAR_CONSTANT(a,b,c,d) \
    ((int32_t)((a) | (b) << 8 | (c) << 16 | (d) << 24))

enum {
    EXT_JPG = MULTICHAR_CONSTANT_L('.','j','p','g'),
```

```

EXT_PNG = MULTICHAR_CONSTANT_L('.','p','n','g'),
EXT_HTM = MULTICHAR_CONSTANT_L('.','h','t','m'),
EXT_CSS = MULTICHAR_CONSTANT_L('.','c','s','s'),
EXT_TXT = MULTICHAR_CONSTANT_L('.','t','x','t'),
EXT_JS = MULTICHAR_CONSTANT_L('.','j','s',0),
} lwan_mime_ext_t;

const char* lwan_determine_mime_type_for_file_name(char *file_name)
{
    char *last_dot = strrchr(file_name, '.');
    if (UNLIKELY(!last_dot))
        goto fallback;

    STRING_SWITCH_L(last_dot) {
    case EXT_CSS: return "text/css";
    case EXT_HTM: return "text/html";
    case EXT_JPG: return "image/jpeg";
    case EXT_JS: return "application/javascript";
    case EXT_PNG: return "image/png";
    case EXT_TXT: return "text/plain";
    }

fallback:
    return "application/octet-stream";
}

```

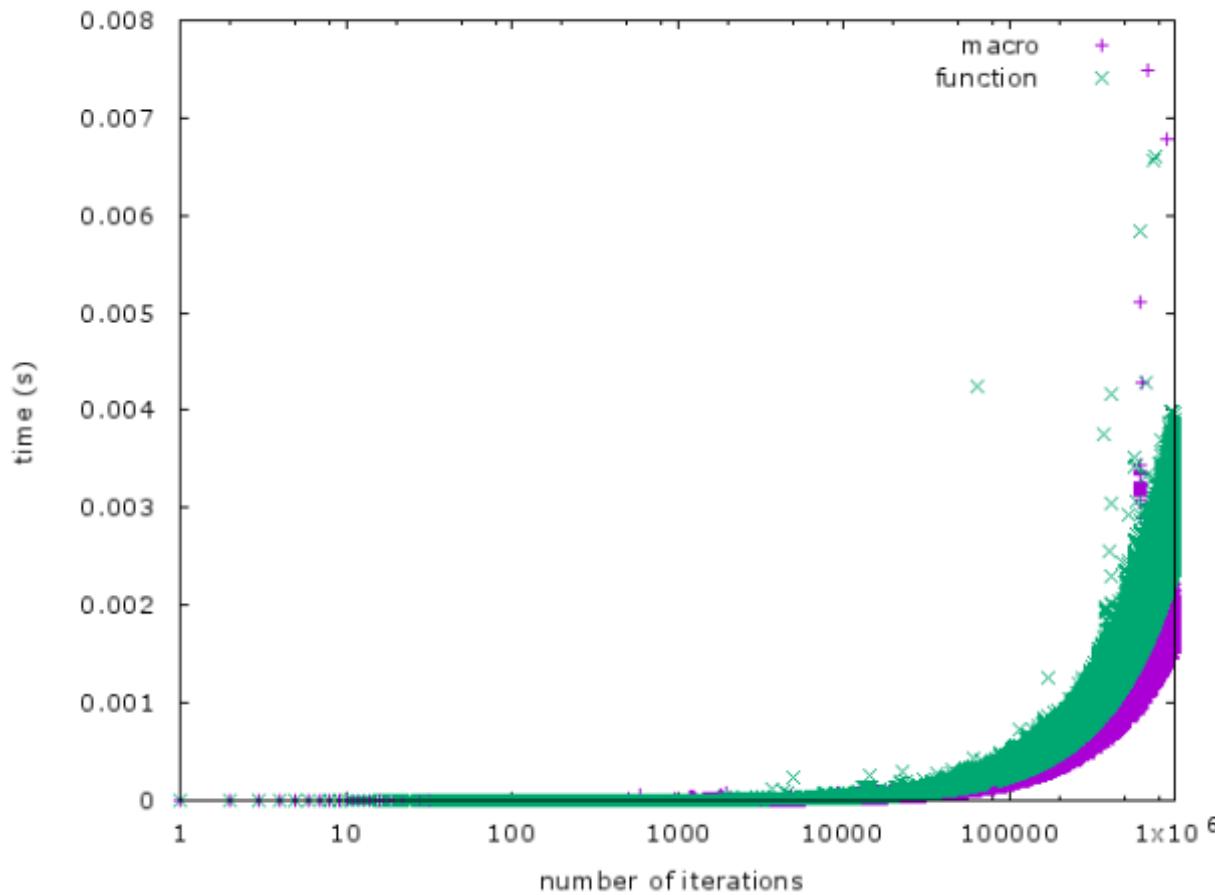
- More on string switch in C

應用: Linked List 的各式變種

- 為何 Linux 核心採用 macro 來實作 linked list ?一般的 function call 有何成本 ?
macro 在編譯時期會被編譯器展開成實際的程式碼，這樣做的好處是不依賴編譯器最佳化。

在進行函式呼叫時，我們除了需要把參數推入特定的暫存器或是堆疊，還要儲存目前暫存器的值到堆疊。在函式呼叫數量少的狀況，影響不顯著，但隨著數量增長，就會導致程式運作比用 macro 實作時慢。

實際執行一段簡單的程式碼，並用 gnuplot 顯示使用 function call 與 macro 的執行時間差異。



Linux 裡頭許多地方都會使用到 linked list 資料結構，因此各種 linked list 的基本操作也會被大量使用。如上段所述，這在 function call 和 macro 實作的效能差異會在量大時顯示出來，這也是為什麼 Linux 會採用 macro 來實作 linked list 的原因之一。

Linux 應用 linked list 在哪些場合？

使用 linked list 最大的好處是程式開發者不需要在撰寫程式之際就決定資料的長度。應用案例：

- Process Management
- inode
- Network File System (NFS)

linux/sched.h

說到 process management，最常提到的應該就是 `task_struct` 這個結構。

定義於 [sched.h](#)，光是定義整個結構就超過 620 行。

節錄在 `task_struct` 裡頭紀錄 parent/child 的部份解釋

```
753  /*
754   * Pointers to the (original) parent process, youngest child, younger sibling
755   * older sibling, respectively. (p->father can be replaced with
756   * p->real_parent->pid)
757   */
758
759  /* Real parent process: */
760  struct task_struct __rcu      *real_parent;
761
762  /* Recipient of SIGCHLD, wait4() reports: */
763  struct task_struct __rcu      *parent;
764
765  /*
766   * Children/sibling form the list of natural children:
767   */
768  struct list_head           children;
769  struct list_head           sibling;
770  struct task_struct        *group_leader;
```

在 `real_parent` 的前面有一個 `__rcu` 的巨集，全稱叫作 [Read-copy-update](#)，他可以允許動態分配位址的資料結構（e.g. linked list）在讀取資料時不需要把整個結構用 mutex 保護。

延伸閱讀: [Linux 核心設計: RCU 同步機制](#)

那麼 `real_parent` 和 `parent` 的差別在哪裡？

根據上方的註解，我們可以知道 `real_parent` 肯定是整個結構的父親，而 `parent` 則是 `SIGCHLD` 還有 `wait4()` 回傳結果物件。

查看 `man 7 signal` 得到 `SIGCHLD Child stopped or terminated` 可以知道 `SIGCHLD` 是一個 child 暫停或是終止時會發出的 signal。

`wait4()` 是 BSD 風格的函式，和他相近的另一個規範在 POSIX 裡的函式是 `waitpid()`，用途是 block 並等待 child 的狀態改變。

inode

inode (index node) 是 Unix 家族或類似作業系統中，負責處理檔案系統的資料結構。

我們可以看到 `fs.h` 裡頭的 `inode` 結構也有 `list_head` 的蹤影

```
668 struct list_head i_lru;          /* inode LRU list */
669 struct list_head i_sb_list;
670 struct list_head i_wb_list;       /* backing dev writeback list */
```

- 延伸閱讀: [Linux 核心設計: 檔案系統概念及實作手法](#)

- 「靜態」的 linked list

考慮以下實作:

```
#include <stdio.h>

#define cons(x, y) (struct llist[]){x, y}
struct llist { int val; struct llist *next; };

int main() {
    struct llist *list = cons(9, cons(5, cons(4, cons(7, NULL))));
    struct llist *p = list;
    for (; p; p = p->next)
        printf("%d", p->val);
    printf("\n");
    return 0;
}
```

以下分析其原理:

```
#define cons(x, y) (struct llist[]){x, y}
```

- 利用 **compound literal** 建立 static linked list
- 可針對未知大小的 array/struct 初始化，也可以直接對特定位置的 array 或 struct 的內容賦值
- compound literal 可以做 lvalue，可以傳特定型別或自己定義的 struct

C99 [6.5.2.5] **Compound literals**

- The type name shall specify an object type or an array of unknown size, but not a variable length array type.
- A postfix expression that consists of a parenthesized type name followed by a braceenclosed list of initializers is a compound literal. It provides an unnamed object whose value is given by the initializer list.
- If the type name specifies an array of unknown size, the size is determined by the initializer list as specified in 6.7.8, and the type of the compound literal is that of the completed array type. Otherwise (when the type name specifies an object type), the type of the compound literal is that specified by the type name. In either case, the result is an lvalue.

將上述程式碼的 `cons(9, cons(5, cons(4, cons(7, NULL))))` 展開如下:

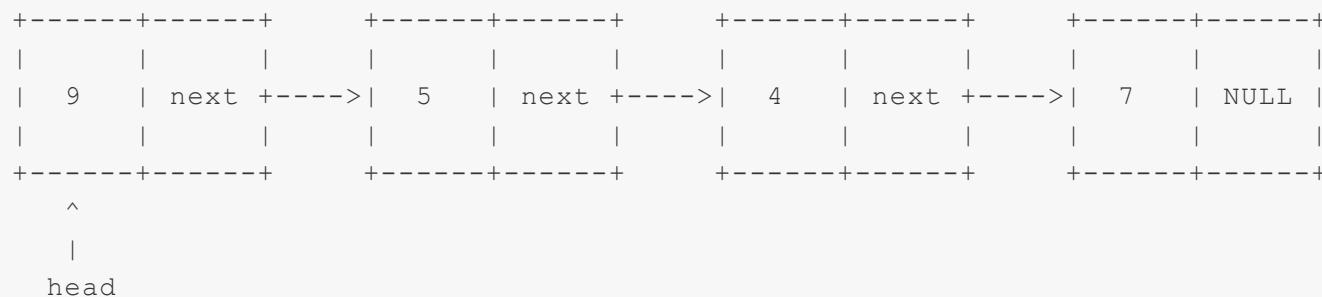
```
// 展開 cons(9, cons(5, cons(4, cons(7, NULL))))
llist.val = 9;
llist.next = &cons(5, cons(4, cons(7, NULL)));
```

```
// 展開 cons(5, cons(4, cons(7, NULL)))
llist.val = 5;
llist.next = &cons(4, cons(7, NULL))

// 展開 cons(4, cons(7, NULL))
llist.val = 4;
llist.next = &cons(7, NULL);

//展開 cons(7, NULL)
llist.val = 7;
llist.next = NULL;
```

便會得到如下的 linked list



從 `cons(7, NULL)` 往外面拆開會比較容易理解指標的方向。因此最終程式輸出結果會是 `9547`

延伸閱讀:

- [Doubly-Linked Lists in C](#)
- [linked list 和非連續記憶體操作](#)

應用: Unit Test

Unity: Unit Test for C
source: [unity](#)
File: [unity/unity_fixture.h](#)

Google Test

Google Mock : an extension to Google Test for writing and using C++ mock classes.

File: [mock/gmock-generated-actions.h](#)

metaclass : In object-oriented programming, a metaclass is a class whose instances are classes. 產生模板的模板

Mock Class: 給定一個表示式，產生對應的 class 和程式碼

應用: Object Model

[ObjectC](#): use as a superset of the C language adding a lot of modern concepts missing in C

Files

- [inc/ObjectC/std/Object.h](#)
- [inc/ObjectC/language/new_delete.h](#)
- [inc/ObjectC/language/type.h](#)

C: exception jmp => setjmp + longjmp

應用: Exception Handling

[ExtendedC](#) library extends the C programming language through complex macros and other tricks that increase productivity, safety, and code reuse without needing to use a higher-level language such as C++, Objective-C, or D.

應用: ADT

[pearlDb](#): A Lightweight Durable HTTP Key-Value Pair Database in C

- 內含 [Klib](#): a Generic Library in C
- Ksort: 在標頭檔中直接展開，省去 function call 的成本，加速資料庫處理的效能。

待整理

- [Metalang99](#): 架構在 C99 前置處理器之上，打造出 recursion, algebraic data types, collections, currying, natural numbers, functional composition 等特徵
- [Macros on Steroids, Or: How Can Pure C Benefit From Metaprogramming](#)
- [A list of awesome C preprocessor stuff](#)
- [Type-Safe Printf For C](#): it uses macro magic, compound literals, and `_Generic` to take `printf()` to the next level: type-safe printing, printing into compound literal char arrays, easy UTF-8, -16, and -32, with good error handling.