C++ Style Guide for SLAM

Header Files

1. Self-contained Headers

> **標頭檔應該能夠獨自被編譯（self-contained），以 `.h` 結尾。至於用來插入其他檔案的文件，說到底它們並不是標頭檔，所以應以 `.inc` 結尾。**

所有標頭檔要能夠 self-contained。換言之，使用者和重構工具不需要為了使用一個標頭檔而引入額外更多的標頭檔。特別是，一個標頭檔應該要有 header guards。

如果 `.h` 文件宣告了一個 templates 或內聯 (inline) 函式，那他的 declaration 與 definition 必須在同一個 `.h` 中，否則程式可能會在構建中連結失敗。有個例外：如果某函式樣板為所有相關模板參數顯式實例化，或本身就是類別的一個私有成員，那麼它就只能定義在實例化該模板的 `.cc` 文件裡。

只有少數的例外，一個標頭檔不是自我滿足的而是用來安插到程式碼某處裡。例如某些文件會被重複的 include 或是文件內容實際上是特定平台（platform-specific）擴展部分。這些文件就要用 .inc 文件擴展名。

2. The #define Guard

> **所有標頭檔都應該使用 `#define` 防止標頭檔被多次引入。建議的命名格式為 `<PROJECT>_<PATH>_<FILE>_H_`**

為保證唯一性，標頭檔的命名應該依據所在專案的完整路徑。例如：專案 foo 中的標頭檔 foo/src/bar/baz.h 可按如下方式保護：

```
#ifndef FOO_BAR_BAZ_H_
#define FOO_BAR_BAZ_H_
…
#endif // FOO_BAR_BAZ_H_
```

3. Include What You Use

> **如果標頭檔參考了在其他地方定義的 symbol，那必須直接 include 該 header file，否則不應該做多餘的 include。任何情況都不應依賴過渡引入(transitive inclusions)。**

顧名思義，只應引入有用到的 symbol。

並且，我們不推薦使用過渡引入。這使得我們可以很輕易的刪除不再使用的 header file 並且不用擔心會導致 clients 編譯失敗。反之，這代表當 `foo.cc` 需要使用 `bar.h` 定義的 symbol 時必須直接引入，儘管 `foo.h` 已經引入過。

4. Forward Declarations

> **避免使用前置宣告。反之，直接引入需要的標頭檔即可。**

Definition:

前置宣告是不提供與之關連的定義下，宣告一個類別、函式或是樣板。

```
// In a C++ source file:
class B;
void FuncInB();
extern int variable_in_b;
ABSL_DECLARE_FLAG(flag_in_b);
```

Pros:

- 由於 #include 會強制編譯器開啟更多的檔案與處理更多的輸入，利用前置宣告減少 #include 可以減少編譯時間。
- 越多的 #include 代表程式碼更可能因為相依的標頭檔更動而被重新編譯，使用前置宣告可以節省不必要的重新編譯。

Cons:

- 前置宣告可能隱藏掉與標頭檔間的相依關係，導致當標頭檔改變時，相依的程式碼沒有被重新編譯。
- 使用前置宣告可能會使某些 IDE 自動工具難以找到其定義。
- 前置宣告可能在函式庫進行改動時發生編譯錯誤。例如函式庫開發者放寬了某個參數類型、替樣板增加預設參數或更改命名空間等等。
- 前置宣告來自 std:: 命名空間的 symbols 會導致未定義行為 (undefined behavior)。
- 難以抉擇是要使用前置宣告或是引入完整得標頭檔。在某些狀況下，使用前置宣告替換掉 #include 可能意外的修改了程式碼的意圖。如下，若 #include 被替換成 B 和 D 的前置宣告，test() 會呼叫到 f(void*)。

```
// b.h:
struct B {};
struct D : B {};

// good_user.cc:
#include "b.h"
void f(B*);
void f(void*);
void test(D* x) { f(x); }  // Calls f(B*)
```

- 使用前置宣告多個 symbols 可能暴露了比直接引入標頭檔更多的訊息。
- 為了使用前置宣告而修改程式碼（例如：使用指標成員而不是物件成員) 可能會導致程式運作較為緩慢或是更加的複雜。

Decision:

在任何狀況下避免使用前置宣告。

5. Inline Functions

> **只有當函式非常的短，例如只有 10 行甚至更少的時候，才將其定義為內聯函式。**

Definition:

當函式被宣告為內聯函式之後，代表你允許編譯器將其展開在該函式被呼叫的位置，而不是原來的函式呼叫機制進行。

Pros:

當函式主體比較小的時候，內聯該函式可以產生更有效率目標程式碼 (object code)。對於存取函式 (accessors)、賦值函式 (mutators) 以及其它函式體比較短或性能關鍵的函式，可以依據需求將其轉為內聯函式。

Cons:

濫用內聯反而會導致程式變慢。內聯可能使目標程式碼變大或變小，這取決於內聯函式主體的大小。一個非常短小的存取函式被內聯通常會減少目標程式碼的大小， 但內聯一個相當大的函式將戲劇性的增加目標程式碼大小。現代的處理器 (CPU) 具備有指令緩存 (instruction cache)，執行小巧的程式碼往往執行更快。

Decision:

一個較為合理的經驗準則是，不要內聯超過 10 行的函式。謹慎對待解構子, 解構子往往比其表面看起來要更長, 因為有隱含的成員和父類別解構子被呼叫！

另一個實用的經驗準則: 內聯那些包含 loop 或 switch 語句的函式常常是得不償失的 (除非在大多數情況下, 這些 loop 或 switch 語句從不被執行)。

要注意的是，既使函式即使宣告為內聯，也不一定會被編譯器內聯。例如虛函式 (virtual) 和遞迴函式 (recursive) 就不會被正常內聯。通常, 遞迴函式不應該宣告成內聯函式。虛函式內聯的主要原因則是想把它的函式主體放在類別的定義內， 可能式為了方便，或是當作文件描述其行為。例如存取函式或賦值函式。

6. Names and Order of Includes

> **使用以下標準的標頭檔引入順序可增強可讀性，同時避免隱藏相依性：相關標頭檔 > C 函式庫 > C++ 函式庫 > 其他函式庫的 .h > 專案內的 .h。**

專案內的標頭檔應按照專案目錄樹結構排列，避免使用 UNIX 特殊的目錄捷徑 . (當前目錄) 或 .. (上層目錄)。例如： `htc-awesome-project/src/base/logging.h` 應該按如下方式引入:

```
#include "base/logging.h"
```

另一個例子是，若 `dir/foo.cc` 或 `dir/foo_test.cc` 的主要作用是實作或測試 `dir2/foo2.h` 的功能， `foo.cc` 中引入標頭檔的次序應如下：

1. `"dir2/foo2.h"`
2. A blank line
3. C 系統文件，e.g.，`<unistd.h>`，`<stdlib.h>`
4. A blank line
5. C++ 系統文件，e.g.，`<algorithm>`，`<cstddef>`
6. A blank line
7. 其他函式庫的 `.h` 文件
8. A blank line
9. 此專案內 `.h` 文件

標頭檔的順序在依照類別分類後，同類別的引入順序則應該依照按字母順序排列。

使用這種排序方式，若 `dir2/foo2.h` 忽略了任何需要的標頭檔，在編譯 `dir/foo.cc` 或 `dir/foo_test.cc` 就會發生錯誤。因此這個規則可以確這些功能的保開發者可以在第一時間就發現錯誤。

舉例來說， `htc-awesome-project/src/foo/internal/fooserver.cc` 的引入次序如下：

```
#include "foo/public/fooserver.h"

#include <sys/types.h>
#include <unistd.h>
#include <hash_map>
#include <vector>

#include "base/basictypes.h"
#include "base/commandlineflags.h"
#include "foo/public/bar.h"
```

例外：

有時，平台特定（system-specific）的程式碼需要依據條件被引入（conditional includes），這些程式碼可以放到其它的 includes 之後。當然，盡量讓你的平台特定程式碼小 (small) 且集中 (localized)，例如：

```
#include "foo/public/fooserver.h"

#include "base/port.h"  // For LANG_CXX11.

#ifdef LANG_CXX11
#include <initializer_list>
#endif  // LANG_CXX11
```

Scoping

7. Namespaces

> **除了少數的例外，都建議使用把程式碼放在命名空間內。一個具名的命名空間應該擁有唯一的名字，其名稱可基於專案名稱，甚至是相對路徑。而在 .cc 文件內，使用匿名的命名空間是推薦的，但禁止使用 using 指示（using-directives）和內聯命名空間（inline namespaces）。**

Definition:

命名空間將全域作用域細分為獨立的，具名的作用域可有效防止全域作用域的命名衝突。

Pros:

命名空間可以在大型專案內避免名稱衝突，同時又可以讓多數的程式碼有合理簡短的名稱。

舉例來說, 兩個不同專案的全域作用域都有一個類別 `Foo` ，這樣在編譯或運行時期會造成衝突。如果每個專案將程式碼置於不同命名空間中， `project1::Foo` 和 `project2::Foo` 在專案中就可以被視為不同的 symbols 而不會發生衝突。兩個類別在各自的命名空間中，也可以繼續使用 `Foo` 而不需要前綴命名空間。

內聯命名空間會自動把內部的標識符放到外層作用域，比如：

```
namespace outer {
inline namespace inner {
  void foo();
}  // namespace inner
}  // namespace outer
```

`outer::inner::foo()` 與 `outer::foo()` 彼此可以互換使用。內聯命名空間主要用來保持跨版本的 ABI 相容性。

Cons:

命名空間可能造成疑惑，因為它增加了識別一個名稱所代表的意涵的難度。例如： `Foo` 是命名空間或是一個類別。

內聯命名空間更是容易令人疑惑，因為它並不完全符合命名空間的定義；內聯命名空間只在大型版本控制裡會被使用到。

在標頭檔中使用匿名命名空間容易導致違背 C++ 的唯一定義原則 (One Definition Rule (ODR))。

在某些狀況中，經常會需要重複的使用完整 (fully-qualified) 的名稱來參考某些 symbols。對於多層巢狀的命名空間，這會增加許多混亂。

Decision:

使用命名空間如下：

- 遵循 Namespace 命名規則
- f
- Namespace 應包含整個 source file，並且排在來自其他 namespaces 的 includes 和前置宣告之後

```
// In the .h file
namespace mynamespace {

// All declarations are within the namespace scope.
// Notice the lack of indentation.
class MyClass {
public:
...
void Foo();
};

}  // namespace mynamespace


// In the .cc file
namespace mynamespace {

// Definition of functions is within scope of the namespace.
void MyClass::Foo() {
...
}

}  // namespace mynamespace
```

更複雜的例子：

```
#include "a.h"

ABSL_FLAG(bool, someflag, false, "a flag");

namespace mynamespace {

using ::foo::Bar;

...code for mynamespace...    // Code goes against the left margin.

}  // namespace mynamespace
```

- 禁止在 `std` 命名空間中定義東西
- 禁止使用 using-directives，這會污染命名空間。

```
// Forbidden -- This pollutes the namespace.
using namespace foo;
```

- 禁止使用內聯命名空間
- 禁止在 `.h` 中使用命名空間別名 (Namespace aliases)，但在 `.cc` 中允許。因為若在 `.h` 將會成為 API 的一部份洩露給所有人。

```
// Shorten access to some commonly used names in .cc files.
namespace baz = ::foo::bar::baz;
```

8. Internal Linkage

> **當 `.cc` 檔案中的定義不需要在該檔案之外引用時，將它們放置在匿名命名空間來賦予它們內部連結 (Internal Linkage)。 禁止在 `.h` 檔案中使用。**

Definition:

所有宣告可以放置在匿名命名空間來賦予他內部連結。同樣的，我們也可透過為 function 與 variables 加上 `static` 來賦予其性質。一旦賦予內部連結性質，所有該檔案以外的地方均無法參考。

Decision:

建議將任何不需給外部參考的東西放置在匿名空間中，這裡不推薦使用 `static` 的原因在於 `static` 在不同地方往往含意不同，這種寫法易造成混淆。禁止在 `.h` 檔案中使用內部連結。

```
namespace {
int i = 20;
}

int main(int, char**) {
  std::cout << "i: " << ::i << std::endl;
}
```

9. Nonmember, Static Member, and Global Functions

> **建議將非成員函式放置在命名空間中，盡量不要使用完全的全域函式。建議利用命名空間來放置相關的多個函式，而不是全部放置在類別中並宣告成 `static` 。類別的靜態方法一般來說要和類別的實例或類別的靜態資料有緊密的關連。**

Pros:

某些情況下，非成員函式和靜態成員函式是非常有用的。將非成員函式放在命名空間內可避免對於全域作用域污染。

Cons:

為非成員函式和靜態成員函式準備一個新的類別可能更有意義，特別是它們需要存取外部資源或式有大量的相依性關係時。

Decision:

有時候定義一個不綁定特定類別實例的函式是有用的，甚至是必要的。這樣的函式可以被定義成靜態成員或是非成員函式。非成員函式不應該依賴於外部變數，且應該總是放置於某個命名空間內。相比單純為了封裝不共享任何靜態數據的靜態成員函式而創建一個類別，不如之直接使用 Namespaces。例如對於 myproject/foo_bar.h 標頭擋來說，可以這樣寫。

```
namespace myproject {
namespace foo_bar {
void Function1();
void Function2();
}
}
```

而不是

```
// Forbidden
namespace myproject {
class FooBar {
 public:
  static void Function1();
  static void Function2();
};
}
```

如果你必須定義非成員函式，又只是在 `.cc` 文件中使用它，則可使用 Internal Linkage 限定其作用域。

10. Local Variables

> **盡可能將函式內的變數的作用域最小化，並在變量宣告時進行初始化。**

C++ 允許在函式內的任何位置宣告變數。我們鼓勵在盡可能小的作用域中宣告變量，並且離第一次使用的地方越近越好。這會讓閱讀者更容易找到變數宣告的位置、宣告的類型和初始值。要注意，應該該宣告時直接初始化變數，而不要先代宣告再後賦值, 例如：

```
int i;
i = f();      // Bad -- initialization separate from declaration.


int i = f();  // Good -- declaration has initialization.


int jobs = NumJobs();
// More code...
f(jobs);      // Bad -- declaration separate from use.


int jobs = NumJobs();
f(jobs);      // Good -- declaration immediately (or closely) followed by use.


std::vector<int> v = {1, 2};  // Good -- v starts initialized.
```

在 `if` 、 `while` 和 `for` 陳述句需要的變數一般都會宣告在這些陳述句中，也就是這些變數會存活於這些作用域內。例如：

```
while (const char* p = strchr(str, '/')) str = p + 1;
```

如果變數是一個物件，每次進入作用域時其建構子都會被呼叫，每次離開作用域時其解構子都會被呼叫。

```
// Inefficient implementation:
for (int i = 0; i < 1000000; ++i) {
  Foo f;  // My ctor and dtor get called 1000000 times each.
  f.DoSomething(i);
}
```

在循環作用域外面宣告這類型的變數可能更加的有效率。

```
Foo f;  // My ctor and dtor get called once each.
for (int i = 0; i < 1000000; ++i) {
  f.DoSomething(i);
}
```

11. Static and Global Variables

> **除非物件是 trivially destructible，不然禁止使用具有 static storage duration 的物件。Static function-local variables 可以使用動態初始化 (dynamic initialization)。 不鼓勵對 namespace 內的變數或者 static class member variables 或進行動態初始化。**
> **作為經驗法則：一個宣告為 `constexpr` 的變數、POD (Plain Old Data) 如 `int` `char` `float` `raw pointer` 、POD array/struct/class ，可以滿足上述要求。**

- trivially destructible: 基本上指 POD。
- static storage duration: 物件的生命週期(lifetime)是從程式開始執行的時候開始，程式結束之後才會被釋放。
- dynamic initialization: 初始值無法在 compile-time 得知，會在 runtime 時計算。

Definition:

每個物體都有一個 storage duration，這與其生命週期相關。 具有 static storage duration 的物件從初始化開始一直持續到程式結束。 此類物件在命名空間範圍內（“全域性變數”）作為變數出現，或作為 static data members of classes，或作為 static function-local variables 出現。Static function-local variables 會在首次呼叫該函式時初始化；而其他類型皆在程式啟動時初始化。 所有具有static storage duration 的物件都會在程式退出時被銷毀（注意：發生在 unjoined threads 被終止之前）。

Dynamic initialization，代表在初始化期間會執行 non-trivial 運算（例如，allocates memory, 變數由 current PID 初始化），也因此無法在 complie-time 得知，反之則是 Static initialization。 Static initialization 總是發生在具有 static storage duration 的物件身上，動態初始化會發生在 runtime。

Pros:

Global and static variables 對多數應用程式非常有用：named constants, auxiliary data structures internal to some translation unit, command-line flags, logging, registration mechanisms, background infrastructure, etc.。

Cons:

使用 dynamic initialization 或具有 non-trivial destructors 的全域性和靜態變數很容易導致難以找到的錯誤。 原因在於，大型專案很難控制個單元的連結順序，建構子、解構子和初始化的順序在 C++ 中規範並不完整，導致每次編譯會產生不同的結果。 當一個靜態變數初始化時使用另一個變數，這可能會導致物件在其生命週期開始前（或其生命週期結束後）被訪問。 此外，當程式啟動 unjoined threads 並且未在結束前 join，這些執行緒可能會在其生命週期結束後嘗試訪問物件。

Decision:

- **Decision on destruction**
  當 destructors 為 trivial，它們的執行完全不受順序約束（它們實際上不會“執行”）；否則，我們將面臨在物件生命週期結束後訪問物件的風險。 因此，我們只允許具有 static storage duration 的物件，前提是它們是 trivially destructible。 如 POD， 標有 `constexpr` 的變數。

```
const int kNum = 10;  // Allowed

struct X { int n; };
const X kX[] = {{1}, {2}, {3}};  // Allowed

void foo() {
  static const char* const kMessages[] = {"hello", "world"};  // Allowed
}

// Allowed: constexpr guarantees trivial destructor.
constexpr std::array<int, 3> kArray = {1, 2, 3};


// bad: non-trivial destructor
const std::string kFoo = "foo";

// Bad for the same reason, even though kBar is a reference (the
// rule also applies to lifetime-extended temporary objects).
const std::string& kBar = StrCat("a", "b", "c");

void bar() {
  // Bad: non-trivial destructor.
  static std::map<int, int> kData = {{1, 0}, {2, 0}, {3, 0}};
}
```

  請注意，references 不是物件，雖然因此不受 destructibility 的限制，但還是受 dynamic initialization 影響。唯一的例外為： `static T& t = *new T` ，這種寫法可被允許。
- **Decision on initialization**
  初始化更為複雜，端看 constructors 的設計，考慮以下全域變數的寫法：

```
int n = 5;    // Fine
int m = f();  // ? (Depends on f)
Foo x;        // ? (Depends on Foo::Foo)
Bar y = g();  // ? (Depends on g and on Bar::Bar)
```

  若使用 constant initialization，則可被允許：

```
struct Foo { constexpr Foo(int) {} };

int n = 5;  // Fine, 5 is a constant expression.
Foo x(2);   // Fine, 2 is a constant expression and the chosen constructor is const
Foo a[] = { Foo(1), Foo(2), Foo(3) };  // Fine
```

任何沒有如此標記的 non-local static storage duration variable 都應推定為具有動態初始化，並非常仔細地審查。
相反的，以下不被允許：

```
// Some declarations used below.
time_t time(time_t*);      // Not constexpr!
int f();                   // Not constexpr!
struct Bar { Bar() {} };

// Problematic initializations.
time_t m = time(nullptr);  // Initializing expression not a constant expression.
Foo y(f());                // Ditto
Bar b;                     // Chosen constructor Bar::Bar() not constexpr.
```

- **Common patterns**
  以下將討論各種需要設為全域或靜態變數的情境：
  - **Global `string`**：若要使用，考慮宣告成 `constexpr` 的 `string_view` （string_view 具有 constexpr constructor 和 trivial destructor），`char[]`，`char*`。
  - **Maps, sets, and other dynamic containers**：任何 dynamic contianers 都不具有 trivial destructor，因此不被允許使用。請考慮用 array of array 或者 array of pair 來取代 （當然，array 裡面也要是 POD）。宣告時可考慮將值排序，以便進行搜尋時使用 binary search 增加效率。
  - **Smart pointers**：smart pointers 不具有 trivial destructor 因此不被允許。若要使用 pointer，直接宣告一個 `raw pointer` 即可。
  - **Static variables of custom types**：若要使用，則需定義對應的 `constexpr` constructor，並且需具有 trivial destructor。
  - 若上述方式都無法滿足需求，還有一種特殊方式可以動態宣告，考慮使用 function-local static pointer / reference:

    ```
    T& GetT() {
      static const auto& impl = *new T(args...);
      retrun impl;
    }
    ```

12. `thread_local` Variables

> **所有未在 function 內宣告的 `thread_local` 變數，必須宣告為 `constexpr` 。任何 thread-local data 請偏好使用 `thread_local` 。**

Definition:

Pros:

Cons:

Decision:

Classes

類別是 C++ 中程式碼的基本單元。想當然爾, 在程式中類別將被廣泛使用。本節列舉了在撰寫一個類別時該做的和不該做的事項.

13. Doing Work in Constructors

> 不要在建構子中呼叫虛函式 (virtual function)，也不要做任何有失敗可能的運算.

Definition:

在建構子體中進行初始化操作.

Pros:

- 無須擔心此類別是否已經初始化.
- 物件由建構子初始化可賦予 `const` ，也可以方便使用於 standard containers 或者演算法中.

Cons:

- 如果在建構子內呼叫了自身的虛函式, 這類呼叫是不會重定向 (dispatched) 到子類的虛函式實作. 即使當前沒有子類化實作, 將來仍是隱患.
- 建構子中難以報錯, 或使用例外.
- 建構失敗會造成對象進入不確定狀態．或許可使用類似 `IsValid()` 的機制去做狀態檢查，但這不具強制性也很容忘記使用．
- 如果有人創建該類型的全域變數 (雖然違背了上節提到的規則), 建構子將先 `main()` 一步被呼叫, 有可能破壞建構函式中暗含的假設條件. 例如,

`gflags <http://code.google.com/p/google-gflags/>` _ 尚未初始化.

Decision:

建構子不得呼叫虛函式, 或嘗試報告一個非致命錯誤. 如果對象需要進行有意義的 (non-trivial) 初始化, 考慮使用明確的 `Init()` 方法或使用工廠模式.

14. Implicit Conversions

> 對單個參數的建構子使用 C++ 關鍵字 `explicit` .

Definition:

通常, 如果建構子只有一個參數, 可看成是一種隱式轉換. 打個比方, 如果你定義了 `Foo::Foo(string name)` , 接著把一個字符串傳給一個以 `Foo` 對象為參數的函式, 建構函式 `Foo::Foo(string name)` 將被呼叫, 並將該字符串轉換為一個 `Foo` 的臨時對像傳給呼叫函式. 看上去很方便, 但如果你並不希望如此通過轉換生成一個新對象的話, 麻煩也隨之而來. 為避免構造函式被呼叫造成隱式轉換, 可以將其宣告為 `explicit` .

除單參數建構子外, 這一規則也適用於除第一個參數以外的其他參數都具有默認參數的建構函式, 例如 Foo::Foo(string name, int id = 42).

Pros:

無

Cons:

無

Decision:

所有單參數建構子都必須是顯式的. 在類定義中, 將關鍵字 `explicit` 加到單參數建構函式前: `explicit Foo(string name);`

例外: 在極少數情況下, 拷貝建構子可以不宣告成 `explicit` . 作為其它類的透明包裝器的類也是特例之一. 類似的例外情況應在註解中明確說明.

最後, 只有 std::initializer_list 的建構子可以是非 explicit, 以允許你的類型結構可以使用列表初始化的方式進行賦值. 例如:

```
MyType m = {1, 2};
MyType MakeMyType() { return {1, 2}; }
TakeMyType({1, 2});
```

15. Copyable and Movable Types

> dj/

Definition:

Pros:

Cons:

Decision:

16. Structs vs. Classes

> 僅當只有數據時使用 `struct`，其它一概使用 `class`.

Decision:

在 C++ 中 `struct` 和 `class` 關鍵字幾乎含義一樣. 我們為這兩個關鍵字添加我們自己的語義理解, 以便在定義數據類型時選擇合適的關鍵字.

`struct` 用來定義包含僅包含數據的對象, 也可以包含相關的常數, 但除了存取數據成員之外, 沒有別的函式功能. 所有數據皆為 `public`，並且僅允許建構子, 解構子, Operator, 與相關的 helper function. 所有變數與函式應避免引入不變性 (invariants).

如果需要更多的函式功能，`class` 更適合. 如果拿不準, 就用 `class`.

為了和 STL 保持一致, 對於 stateless types 的特性可以不用 `class` 而是使用 `struct`，像是 traits, template metafunctions, etc.

注意: 類和結構體的成員變數使用不同的命名規則.

17. Structs vs. Pairs and Tuples

> 使用 `struct`，而非 `pair` 或者 `tuple`.

Decision:

當使用 `pair` 或者 `tuple` 時對於程式撰寫者有很大的方便性，然而對於閱讀者而言卻造成不便．光看 `.first`，`.second`, 或 `std::get<X>` 無法清楚的知道順序對應的含義，閱讀者必須移至宣告處才能明白．相反的，使用 `struct` 我們可以直接透過命名理解，更加增進閱讀效率．

18. Inheritance

> dj/

Definition:

Pros:

Cons:

Decision:

19. Operator Overloading

> dj/

Definition:

Pros:

Cons:

Decision:

20. Access Control

> 將所有數據成員宣告為 `private`, 除非他是 `constant`. 並根據需要提供相應的存取函式. 命名規則為, 某個名為 `foo_` 的變數, 其取值函式是 `foo()`. 賦值函式是 `set_foo()`. 一般在標頭檔中把存取函式定義成 inline function.

21. Declaration Order

類的訪問控制區段的宣告順序依次為: `public:`, `protected:`, `private:`. 如果某區段沒內容, 不宣告．

每個區段內的宣告按以下順序:

- Types and type aliases (typedef, using, enum, nested structs and classes, and friend types)
- (Optionally, for structs only) non-static data members

- Static constants
- Factory functions
- Constructors and assignment operators
- Destructor
- All other functions (static and non-static member functions, and friend functions)
- All other data members (static and non-static)

不要在類中定義大型 inline function. 通常, 只有那些沒有特別意義或性能要求高, 並且是比較短小的函式才能被定義為 inline function. 更多細節參考 內聯函式.

Functions

22. Inputs and Outputs

23. Write Short Functions

24. Functions Overloading

25. Default Arguments

26. Trailing Return Type Syntax

C++ Features

27. Ownership and Smart Pointers

28. Rvalue References

29. Friends

30. Exceptions

31. noexcept

32. Run-Time Type Information (RTTI)

33. Casting

34. Streams

35. Preincrement and Predecrement

36. Use of const

37. Use of constexpr, constinit, and consteval

38. Integer Types

39. 64-bit Portability

30. Preprocessor Macros

31. 0 and nullptr/NULL

32. sizeof

33. Type Deduction (including auto)

34. Class Template Argument Deduction

35. Designated Initializers

36. Lambda Expressions

37. Template Metaprogramming

38. Concepts and Constraints

39. Boost

40. Other C++ Features

41. Nonstandard Extensions

42. Aliases

43. Switch Statements

Naming

44. General Naming Rules

45. File Names

46. Type Names

47. Variable Names

48. Constant Names

49. Function Names

50. Namespace Names

51. Enumerator Names

52. Macro Names

53. Exceptions to Naming Rules

Comments

54. Comment Style

55. File Comments

56. Struct and Class Comments

57. Function Comments

58. Variable Comments

59. Implementation Comments

60. Punctuation, Spelling, and Grammar

61. TODO Comments