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
- 禁止帶有註釋的多行名稱空間，如以下例子所示。
- 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;


// Shorten access to some commonly used names (in a .h file).
namespace librarian {
namespace impl {  // Internal, not part of the API.
namespace sidetable = ::pipeline_diagnostics::sidetable;
}  // namespace impl

inline void my_inline_function() {
  // namespace alias local to a function (or method).
  namespace baz = ::foo::bar::baz;
  ...
}
}  // namespace librarian
```

- 禁止使用內聯命名空間
- 使用名稱中帶有 "internal" 的 namespace 來記錄 API 使用者不應提及的API 部分。

```
// We shouldn't use this internal name in non-absl code.
using ::absl::container_internal::ImplementationDetail;
```

- 單行巢狀 namespace 宣告是首選，但不是必需的。

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:

變數可以宣告為 `thread_local`：

```
thread_local Foo foo = ...;
```

這種變數實際上是物件的集合，因此當不同的執行緒訪問它時，它們實際上是在訪問不同的物件。 `thread_local` 可看作為單一 thread 範疇內的 static storage duration variables，例如：他可以宣告在 namespace，可以在 function 內，或者為 static class members，但不可作為一般的 class members。

`thread_local` 變數初始化與靜態變數非常相似，只是它們必須為每個執行緒單獨初始化，而不是在程式啟動時初始化一次。 這意味著函式中宣告的 `thread_local` 變數是安全的，但在其他地方宣告的 `thread_local` 變數會受到與靜態變數相同的初始化順序問題（以及更多）。

`thread_local` 變數有一個微妙的銷毀順序問題：在執行緒關閉期間， `thread_local` 變數將按照其初始化的相反順序被銷毀。 如果由任何 `thread_local` 變數的 destructor 中訪問任何已被破壞的 `thread_local` ，將產生難以發現的 use-after-free bug。

Pros:

- Thread-local data 能有效避免 data racing 問題，適合用於 concurrent programming.
- `thread_local` 是唯一一個 c++ 標準支持創建 thread-local data 的方式。

Cons:

- 每當新的 thread 開始， `thread_local` 變數的 constructor 也無可避免的需要被重新運算。
- `thread_local` 變數是一種全域變數，因此也同樣有全域變數的所有缺點（但他會是 thread-safety）。
- 隨著 thread 數量上升， `thread_local` 變數所佔用的記憶體也隨之上升。
- 一般的 data member 不能為 `thread_local` 除非他是 `static` 。
- 有可能發生 use-after-free bug，如上所述。

Decision:

當想要在 namespace 或者 class scope 中宣告 `thread_local` 變數，請使其為一個 `constexpr` ，或者使用 function-local 技巧：

```
constexpr thread_local Foo foo = ...;

// or
Foo& MyThreadLocalFoo() {
  thread_local Foo result = ComplicatedInitialization();
  return result;
}
```

在函式內宣告的 `thread_local` 變數則沒有初始化問題，但也會有潛在的 use-after-free bug 風險。實務上，請考慮使用 trival data types，或者在 destructor 中避免使用自定義的 class，以免訪問到其他的 `thread_local` 變數。

當有使用 thread-local data 的需求，請使用 `thread_local` 取代自定義的機制。

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．
- 建構子無法獲得地址，意味著無法移交至 thread 中運算。

Decision:

建構子不得呼叫虛函式。 若物件初始化過於複雜，有潛在失敗的風險, 考慮使用明確的 `Init()` 方法以便進行錯誤處理，或更進一步使用工廠模式包裝，防止物件未正確初始化：

```
// foo.h
class Foo {
 public:
  // Factory method: creates and returns a Foo.
  // May return null on failure.
  static std::unique_ptr<Foo> Create();

  // Foo is not copyable.
  Foo(const Foo&) = delete;
  Foo& operator=(const Foo&) = delete;

 private:
  // Clients can't invoke the constructor directly.
  Foo();
  void Init();
};

// foo.c
std::unique_ptr<Foo> Foo::Create() {
  // Note that since Foo's constructor is private, we have to use new.
  auto inst = std::make_unique<Foo>();
  inst.Init(); // Ensure proper initialization
  return std::move(inst);
}
```

14. Implicit Conversions

> **不要定義隱式轉換（Implicit conversion）。對單個參數的建構子與 conversion operator (e.g., `operator bool()`) 使用 C++ 關鍵字 `explicit`。**

Definition:

隱式轉換允許將 source type 轉換為 destination type，如將 `int` 傳入接受 `double` 參數的函式。

除此之外，透過 conversion operator，C++ 允許自定義的隱式轉換。而當自定義的 constructor 為單參數的建構子時（或者只有一個參數沒有預設值，e.g., `Foo::Foo(string name, int id = 42)`），隱式轉換也會發生。

`explicit` 可使用在建構子與 conversion operator，確保了 destination type 必須與 source type 一致。此效用不但防止隱式轉換，同時也作用於 list initialization。如下例：

```
class Foo {
  explicit Foo(int x, double y);
  ...
};

void Func(Foo f);


Func({42, 3.14});  // Error
```

Pros:

- 隱式轉換可以增加便利性，不必做明確的轉型（type casting）。
- 隱式轉換可以成為 overloading 的簡單替代方案，例如當帶有 `string_view` 引數的函式也可接收 `std::string` 和 `const char*` ，不必再定義重載。
- 列表初始化（list initialization）語法是初始化物件的一種簡潔方式。

Cons:

- 隱式轉換會造成隱藏型別不匹配錯誤，或者使用者不知道將發生何種轉換。
- 隱式轉換可以使程式更難閱讀，特別是在 overloading 的情況下，因為無法判定何種函式被呼叫。
- 單參數的建構函式可能會意外地用作隱式型別轉換，即使不打算這樣做。
- 當單參數構造函式沒有標記為 `explicit` 時，沒有可靠的方法來判斷它是否旨在定義隱式轉換，還是作者只是忘記標記它。
- 隱式轉換可能會導致呼叫模糊性，特別是當有雙向隱式轉換時。 這發生於由兩種提供隱式轉換的型別引起的，也可以是由同時具有隱式建構子和 conversion operator 的型別造成的。
- 如果目標型別是隱式的，特別是如果列表只有一個元素，列表初始化可能會遇到同樣的問題。

Decision:

所有單參數建構子與 conversion operator 都必須是顯式的。但 copy & move constructors 不能為顯式, 因為他們不涉及型別轉換.

例外: 當物件可用不同型別表示，其背後的值相同時，可與你的 leader 討論是否可忽略此規則。

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};` 。

15. Copyable and Movable Types

> **Class 的 public API 必須明確宣告該類是 copyable, move-only, 或者都不是。 如果這些操作對您的型別來說清楚且有意義，請支援 copy 和/或 move。**

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 getter/setter 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: `class` 和 `struct` 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

> **多用組合，少用繼承。 使用繼承時，定義為 `public` 。**

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 `const` 。 並根據需求提供相應的存取函式。**

將變數設為 private 可以有效地防止非預期的修改。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。

Functions

22. Inputs and Outputs

C++函式的輸出是透過 `return` 提供的，有ㄅ時也可透過 output parameters（or in/out parameters）。

盡可能的使用 `return` 而非 output parameters：它們提高了可讀性，並且通常提供相同或更好的效能(現在編譯器都有 RVO 優化，更多請參考 copy elision 的解釋，或者此篇文章：Revisiting output parameters usefulness (in C++))。

請使用 return by value 或者 return by reference。 禁止使用 return by pointer（smart pointer 可被允許，並且可為 `nullptr`）。

Parameters 要麼是 input parameters，要麼是 output parameters，要麼兩者兼具：

- **Input paramters**: 請 pass by const reference/value，禁止使用 optional input，除非他用於 debug（請參考：Avoid Passing Booleans to Functions）。若使用 optional input，請盡可能的使用 `std::optional` 以增加可讀性。
- **Output parameters & Input/Output parameters**: 若有需要使用此，請 pass by reference。禁止使用 optional output，除非他用於 debug。若使用 optional output，請盡可能的使用 `std::optional` 以增加可讀性。

在使用 pass by const reference 時，避免定義 const reference parameter 比 function call 生命週期長的函式，因為 const reference parameter 可以連結到 `rvalue` ，看下面例子:

```
class StringHolder {
public:
  // The input `val` must live as long as this object,
  // not just the call to this constructor
  StringHolder(const string& val) : val_(val) {}

  const string& get() { return val_; }
private:
  const string& val_;
}

//----

StringHolder holder("abc"s); // temporaries bind to const-ref
std::cout << holder.get();  // boom, UB.
// The string temporary has already been destroyed, the reference is dangling.
```

相反，找到一種方法來消除生命週期需求（例如，透過 pass by value）。

排序函式 parameters 時，將所有 input parameters 排在 output parameters 之前。

23. Write Short Functions

> **盡可能寫小而集中的功能，並且遵循：一個函式只做一件事。**

如果一個函式超過大約40行，請考慮是否可以在不損害程式結構的情況下將其分解。

即使您的大函式現在工作正常，在幾個月內修改它的人可能會新增新行為。 這可能會導致難以找到的錯誤。 保持函式的簡短和只做一件事的原則，使其他人更容易閱讀和修改您的程式。**小函式也更容易測試**。

24. Functions Overloading

> **當變體之間沒有語義差異時，您可以重載函式。避免讓使用者呼叫時難以猜測實際上被呼叫的函式。**

Definition:

您可以編寫一個接受 `const std::string&` 的函式，並用另一個接受 `const char*` 的函式重載它。 (然而，在這種情況下，請考慮 `std::string_view` 。)

```
class MyClass {
 public:
  void Analyze(const std::string &text);
  void Analyze(const char *text, size_t textlen);
};
```

Pros:

透過允許同名函式接受不同的引數，overloading 可以使程式更直觀。 它可能需要模板化程式，並且對使用者來說很方便。

基於const或ref資格的超載可能會使實用程式更可用、更高效，或兩者兼而有之。 （有關更多資訊，請參閱TotW 148。）

Cons:

如果函式 overloading 僅針對 parameter type 變化，讀者可能必須瞭解C++的複雜匹配規則才能知道發生了什麼。 如果 derived class 只覆蓋函式的一些 variants，許多人也會對繼承的語義感到困惑。

Decision:

當變體之間沒有語義差異時，您可以重載函式，否則考慮定義新的函式。 這些重載可能因型別、qualifiers 或 parameters 數量而異。如果您可以在 header 中用簡單的 comment 記錄 overload set 中的所有變體，這是一個好跡象，表明它是一個設計良好的 overload set。

25. Default Arguments

> **預設值只被允許用於 non-virtual function，因為這保證始終具有相同的值。 遵循與 Function overloading 相同的限制，如果預設引數獲得的可讀性沒有比以下缺點更有價值，則考慮重載函式。**

Pros:

通常，您有一個使用預設值的函式，但偶爾您想覆蓋預設值。 預設引數允許一種簡單的方法來做到這一點，而無需為罕見的異常定義許多函式。 與重載函式相比，預設引數具有更清晰的語法，樣板更少，並且“必需”和“可選”引數之間的區別更清晰。

Cons:

預設引數是實現 function overloading 的另一種方式，因此所有 function overloading 的缺點 都適用。

Virtual function 呼叫中 parameter 的預設值由目標物件的靜態型別決定，不能保證給定函式的所有 override 都宣告相同的預設值，例如：

```
struct A {
    virtual void display(int i = 5) { std::cout << "Base::" << i << "\n"; }
};
struct B : public A {
    virtual void display(int i = 9) override { std::cout << "Derived::" << i << "\n"; }
};

int main()
{
    A * a = new B();
    a->display(); // Derived::5

    A* aa = new A();
    aa->display(); // Base::5

    B* bb = new B();
    bb->display(); // Derived::9
}
```

預設引數在每個呼叫點重新計算，這可能會使生成的程式膨脹。 使用者可能還期望預設值在宣告時固定，而不是在每次呼叫時變化。

存在預設引數時，函式指標會令人困惑，因為函式 signature 通常與呼叫 signature 不匹配。 新增 function overloading 可以避免這些問題。

Decision:

預設引數在 virtual function 上被禁止，因為在指定的預設值可能不計算為相同值。

在其他一些情況下，預設引數可以提高其函式宣告的可讀性，足以克服上述缺點，因此允許它們。 當有疑慮時，請使用 overloading。

26. Trailing Return Type Syntax

> **僅在典型的語法難以閱讀時才使用 trailing return type syntax。**

Definition:

C++允許兩種不同形式的函式宣告：

```
int foo(int x);
auto foo(int 2) -> int; // trailing return type syntax
```

Pros:

Trailing return type 是顯式指定 lambda表示式 的返回型別的唯一方法。 在某些情況下，編譯器能夠推斷lambda 的返回型別，但並非在所有情況下。 即使編譯器可以自動推斷它，有時明確指定它對讀者來說會更清晰。

有時，在函式的 parameter list 已經宣告候，指定返回型別更容易，也更容易閱讀。 當返回型別取決於模板引數時，情況尤其如此。 例如：

```
template <typename T, typename U>
auto add(T t, U u) -> decltype(t + u);
```

對比

```
template <typename T, typename U>
decltype(declval<T&>() + declval<U&>()) add(T t, U u);
```

Cons:

Trailing return type syntax 相對較新，因此一些讀者可能會不熟悉。

現有的程式庫有大量的函式宣告，這些宣告不會被更改為使用新語法，因此現實的選擇是只使用舊語法或使用兩者的混合。 使用單一版本更適合風格的一致性。

Decision:

在大多數情況下，繼續使用舊的函式宣告樣式，返回型別在函式名稱之前。 僅在需要時（如lambdas）或透過將型別放在函式引數列表後，允許您以更可讀的方式編寫型別時使用 trailing return type。 後一種情況應該很罕見；這在很大程度上是一個相當複雜的模板程式中的問題，在大多數情況下不鼓勵這樣做。

C++ Features

27. Ownership and Smart Pointers

> **對於 dynamically allocated objects，請盡可能使用 fixed ownership，如 `std::unique_ptr` 。 請盡可能使用 smart pointers 來轉移所有權。**

Definition:

"Ownership" 是一種用於管理動態分配的記憶體（和其他資源）的技術。 動態分配物件的所有者是一個 object 或 function，負責確保在不再需要時將其刪除。 所有權有時可以共享，在這種情況下，最後一個所有者通常負責刪除它。 即使所有權沒有共享，也可以進行轉移。

Smart Pointers 是像指標一樣的 class，例如，透過 overloading `*` 和 `->` 運算子。 一些智慧指標型別可用於自動化所有權紀錄，以確保滿足這些職責。 `std::unique_ptr` 是一種智慧指標型別，表示動態分配物件的排他性所有權；當 `std::unique_ptr` 超出 scope 時，該物件將被刪除。 它無法複製，但可以移動以表示所有權轉移。 `std::shared_ptr` 是一種智慧指標型別，表示動態分配物件的共享所有權。 `std::shared_ptrs` 可以複製；物件的所有權在所有副本之間共享，當最後一個std::shared_ptr被銷燬時，該物件將被刪除。

Pros:

- 如果沒有某種所有權邏輯，幾乎不可能管理動態分配的記憶體。
- 轉讓物件的所有權可能比複製更節省資源（如果複製可能的話）。
- 轉讓所有權可能比“借用” pointer 或 reference 更簡單，因為它減少了在兩個使用者之間協調物件生命週期的需要。
- 智慧指標可以透過明確、自我記錄和明確所有權邏輯來提高可讀性。
- 智慧指標可以消除手動所有權紀錄，簡化程式並排除錯誤。
- 對於 const 物件，shared ownership 可以成為 deep copying 的簡單而有效的替代方案。

Cons:

- 所有權必須透過指標（無論是智慧還是普通）來表示和轉移。 Pointer 語義比 value 語義更復雜，特別是在API中：您不僅要擔心所有權，還要擔心別名、生命週期和可變性等問題。
- Value 語義的效能成本往往被高估了，因此所有權轉讓的效能效益可能無法證明改進可讀性和複雜性成本。
- 轉移所有權的API迫使其客戶端進入單個記憶體管理模型。
- 使用智慧指標的程式對 release memory 的位置是不明確的。
- `std::unique_ptr` 使用移動語義表示所有權轉移，這相對較新，可能會讓一些程式設計師感到困惑。

- 共享所有權過於方便使人忽略謹慎的所有權設計，混淆了系統的設計。
- 共享所有權要求在執行時進行明確的 ownership 紀錄，這可能很浪費效能。
- 在某些情況下（例如，相互引用），具有共享所有權的物件可能永遠不會被刪除。
- Smart pointer 不是 pointer 的完美替代品。

Decision:

如果需要 dynamically allocated objects，請偏好使分配它的物件擁有所有權。 如果其他程式需要訪問該物件，請考慮在不轉讓所有權的情況下傳遞 copy，或傳遞 pointer 或 reference。 更喜歡使用 `std::unique_ptr` 來明確所有權轉移。 例如：

```
std::unique_ptr<Foo> FooFactory();
void FooConsumer(std::unique_ptr<Foo> ptr); // move the ownership of object into functi
void FooConsumer(std::unique_ptr<Foo>& ptr); // pass by reference without moving owners
```

不要在沒有充分理由的情況下設計使用 shared ownership object。 其中一個原因是避免昂貴的複製操作，但只有在效能優勢顯著且底層物件不可變（即 `std::shared_ptr<const Foo>` ）時，您才應該這樣做。 如果您確實使用共享所有權，則更喜歡使用 `std::shared_ptr` 。

切勿使用 `std::auto_ptr` 。 相反，使用 `std::unique_ptr` 。

28. Rvalue References

> 僅在下面列出的某些特殊情況下使用右值引用（rvalue reference）。

Definition:

右值引用是一種只能綁定到臨時物件的 reference type。 語法與傳統參考語法相似。 例如， `void f(std::string&& s);` 宣告一個函式，其引數是 `std::string` 的 右值引用。

當 "&&" 應用於函式引數中的不合規模板引數時，適用特殊的模板引數推導規則將被採納。 這種引用被稱為 forwarding reference (aka universal references)。

Pros:

- 定義 move constructor 可以移動值而不是複製它。 例如，如果 `v1` 是一個 `std::vector<std::string>` ，那麼 `auto v2(std::move(v1))` 可能只會導致一些簡單的指標操作，而不是複製大量資料。 在許多情況下，這可能會導致重大的效能提升。
- Rvalue引用使實現可移動但不可複製的型別成為可能，這對沒有合理的複製定義的型別很有用，但您可能仍然希望將它們作為函式引數傳遞，將它們放入容器等。
- `std::move` 是有效利用某些標準庫型別所必需的，例如 `std::unique_ptr` 。

- 使用 Forwarding reference 可以編寫一個通用函式 wrapper，將其引數轉發到另一個函式，無論其引數是否為臨時物件和/或常量。 這被稱為 "perfect forwarding"。

Cons:

- Rvalue引用尚未被廣泛理解。 引用崩潰和轉發引用的特殊扣除規則等規則有點晦澀。
- Rvalue引用經常被濫用。 在引數預計在函式呼叫後具有有效的指定狀態，或者不執行移動操作的簽名中，使用rvalue引用是違反直覺的。

Decision:

不要使用rvalue引用，除非如下：

- 您可以使用它們來定義 move constructor 和 move assignment operators。
- 您可以將 forwarding reference 與 `std::forward` 一起使用，以支援 perfect forwarding。
- 您可以使用它們來定義成對的 overloads，例如一個接受 `Foo&&` ，另一個接受 `const Foo&` 。 通常，首選的解決方案是 pass by value，但一對 overloads的函式有時會產生更好的效能，有時在需要支援各種型別的通用程式中是必要的。 一如既往：如果您為了效能而編寫更復雜的程式，請確保您有證據表明它確實有幫助。

29. Friends

> 我們允許在合理範圍內使用 friend classes 和 function。

`friend` 通常應該在同一檔案中定義，這樣讀者就不必檢視另一個檔案來查詢類的 private member 的用途。 `friend` 的一個常見用途是讓 `FooBuilder` 類成為 `Foo` 的 `friend` ，這樣它就可以正確構建 `Foo` 的內部狀態，而不會向外部暴露這種狀態。 在某些情況下，讓單元測試類成為它所測試的類的 `friend` 是有用的。

`friend` 不應打破一個 class 的封裝邊界。 在某些情況下，當您只想讓一個 class 訪問 private members 時，這比 public members 要好。 然而，大多數的 class 應該僅透過其 public members 與其他 class 互動。

30. Exceptions

> 我們不使用 C++ exceptions，只允許在檔案存取相關的 function 中使用，並且需與您的 manager 討論合理性。

Pros:

- 例外允許更外層的應用程式決定如何處理 deeply nested functions 中“不該發生”的故障，而無需使用模糊和容易出錯的 error code 方式。

- 大多數其他現代語言都使用例外。 在C++中使用它們將使其與Python、Java和其他語言更加一致。
- 一些第三方C++庫使用異常，在內部關閉它們會更難與這些庫整合。
- 例外是 constructor 失敗的唯一途徑。 我們可以用工廠函式或 `Init()` 方法模擬這一點，但這些分別需要 heap allocation 或新的“無效”狀態。
- 例外在測試框架中非常方便。

Cons:

- 當您將 `throw` 語句新增到現有函式中時，您必須檢查其所有傳遞呼叫者。 要麼他們必須至少做出基本的例外安全保證，要麼他們永遠不能 catch the exception，並對因此終止的程式感到滿意。 例如，如果 `f()` 呼叫 `g()` 呼叫 `h()` ，而 `h` throw 一個由 `f` catch的異常， `g` 必須小心不去 catch，否則它可能無法正確清理。
- 更一般地說，例外使程式的控制流難以透過檢視程式來評估：函式可能會返回您意想不到的地方。 這導致了可維護性和除錯困難。 您可以透過一些關於如何以及在哪裡使用例外的規則來最大限度地降低成本，但代價是開發人員需要知道和理解的更多。
- 例外安全需要 RAII 和不同的編碼實踐。 需要大量的努力來簡化正確的exception-safe code。 此外，為了避免要求讀者理解整個呼叫流，exception-safe code 必須避免將改變狀態的邏輯在 commit 階段發生。 這將既有好處也有成本（也許您被迫混淆程式以隔離 commit）。 允許例外將迫使我們始終付出這些代價，即使它們不值得。
- 開啟 exceptions 會將資料新增到生成的每個 binary 中，增加編譯時間（可能略有），並可能增加 address space 壓力。
- 例外的可用性可能會鼓勵開發人員在不合適時 throw 它們，或者在不安全時從它們中恢復。 例如，無效的使用者輸入不應導致 throw exception。 我們需要讓風格指南更長，以記錄這些限制！

Decision:

從表面上看，使用例外的好處大於成本，特別是在新專案中。 然而，對於現有程式，引入例外會對所有依賴程式產生影響。 如果例外可以在新專案之外傳播，那麼將新專案整合到現有的無異常程式中也會成為問題。 由於我們現有的大多數C++程式沒有準備好處理例外，因此採用生成例外的新程式相對困難。

鑑於我們的現有程式不容忍異常，使用異常的成本略高於新專案的成本。 轉換過程會很慢，容易出錯。 我們不認為可用的例外替代方案，如 error codes 和 assertions，會帶來沉重的負擔。

此禁令也適用於異常處理相關功能，如 `std::exception_ptr` 和 `std::nested_exception` 。

在檔案存取相關的功能中 (如 map, anchor)，既有的寫法直接讀取 serialized data 並當資料異常時會使程式 crash。考慮到版本相容的問題，我們通融在檔案讀取相關的 function 使用 exception。

31. noexcept

> 指定 `noexcept` ，當它有用且正確時。

Definition:

`noexcept` 用於指定函式是否會 throw exception。 如果 exception 從標記為 `noexcept` 的函式中回傳，程式將透 `std::terminate` crash。

`noexcept` 運算子會在編譯時檢查，如果宣告不丟擲任何異常，則返回true。

Pros:

- 在某些情況下，將 move constructor 指定為 `noexcept` 會提高效能，例如，如果 `T` 的 move constructor 是 `noexcept` ，則 `std::vector<T>::resize()` 移動而不是複製物件。
- 在函式上指定 `noexcept` 可以在啟用 exception 的環境中觸發編譯器最佳化，例如，如果編譯器知道由於 `noexcept` 而不能丟擲異常，則編譯器不必為 stack-unwinding 生成額外的程式。

Cons:

- 在遵循本指南禁用 exceptions 的專案中，很難確保 `noexcept` 是正確的，甚至很難定義正確性意味著什麼。

Decision:

如果它準確地反映了函式的預期語義，即如果從函式主體內以某種方式 throw exception，那麼它代表了一個致命的錯誤，那麼當它對效能有用時，您可以使用 `noexcept` 。 您可以假設 noexcept on move constructor 具有有意義的效能優勢。 如果您認為在其他功能上指定 `noexcept` 具有顯著的效能益處，請與您的專案主管討論。

32. Run-Time Type Information (RTTI)

> 避免使用 run-time type information (RTTI).

Definition:

RTTI 允許程式設計師在執行時查詢物件的 type。 通常使用 `typeid` 或 `dynamic_cast` 。

Pros:

RTTI（如下所述）不需要修改或重新設計相關 class 層次結構。 有時這種修改是不可行的或不可取的，特別是在廣泛使用或成熟的程式中。

RTTI在某些單元測試中很有用。 例如，它在工廠類的測試中很有用，在這些測試中，測試必須驗證新建立的物件是否具有預期的動態型別。 它在管理物件及其子類之間的關係方面也很有用。

在考慮多個抽象物件時，RTTI很有用。 例如：

```
bool Base::Equal(Base* other) = 0;
bool Derived::Equal(Base* other) {
  Derived* that = dynamic_cast<Derived*>(other);
  if (that == nullptr)
    return false;
  ...
}
```

Cons:

在執行時查詢物件的型別通常意味著設計問題。 需要知道執行時物件的型別通常表明您的類層次結構設計存在缺陷。

不守紀律地使用 RTTI 使程式難以維護。 它可以導致基於型別的決策樹或使 `switch` 語法分散在程式中的各個地方，並且在日後更改時必須檢查所有邏輯。

Decision:

RTTI 有合裡的用途，但容易被濫用，因此您在使用時必須小心。 您可以在單元測試中自由使用它，但在其他程式中儘可能避免使用它。 特別是，在新程式中使用 RTTI 之前要三思而後行。 如果您發現自己需要根據物件的類編寫行為不同的程式，請考慮以下查詢型別的替代方案：

- 虛擬方法是根據特定子類型別執行不同程式路徑的首選方式。 這把工作放在物件本身裡面。
- 如果工作屬於物件之外，請考慮雙重分派（double-dispatch）解決方案，例如訪客設計模式（Vesiter design pattern）。 這允許物件本身之外的程序使用內建型別系統確定類的型別。

當程式的邏輯保證 base class 的實例實際上是特定 derived class 的實例時，則可以在物件上自由使用 `dynamic_cast` 。 在這種情況下，通常可以使用 `static_cast` 作為替代方案。

基於型別的決策樹強烈表明您的程式走錯了軌道:

```
if (typeid(*data) == typeid(D1)) {
  ...
} else if (typeid(*data) == typeid(D2)) {
  ...
} else if (typeid(*data) == typeid(D3)) {
...
```

當新的子類被新增到類層次結構中時，RTTI 的寫法通常會導致錯誤。 此外，當子類的屬性更改時，很難找到和修改所有受影響的程式段。

不要手動實施類似 RTTI 的變通方法。 反對 RTTI 的論點同樣適用於變通方法，如帶有型別標籤的類層次結構。 此外，變通方法掩蓋了你的真實意圖。

33. Casting

> 使用C++風格的轉換，如 `static_cast<float>(double_value)` ，或大括號初始化來轉換算術型別，如 `int64_t y = int64_t{1} << 42` 。 除非轉換無效，否則不要使用像 `(int)x` 這樣的轉換格式。 只有當 `T` 是類型別時，您才能使用 `T(x)` 等轉換格式。

Definition:

C++ 引入了一個與 C 不同的 casting 系統，可以區分不同意圖的 casting。

Pros:

C casting 的問題在於操作的模糊性；有時您正在進行 conversion（例如 `(int)3.5` ），有時您正在進行 cast（例如 `(int)"hello"` ）。 大括號初始化和 C++ casting 通常可以幫助避免這種模糊性。 此外，C++ casting 在搜尋時更清晰可見。

Cons:

C++風格的 casting 語法很冗長。

Decision:

一般來說，不要使用 C casting。 相反，當需要顯式型別轉換時，請使用這些 C++ 風格的轉換。

- 使用大括號初始化來轉換算術型別（例如， `int64_t{x}` ）。 這是最安全的方法，因為如果轉換會導致資訊丟失，程式將無法編譯。 語法也很簡潔。
- 當您需要將 pointer 從子類 explicitly up-cast 到其父類時，或者相反，請使用 `static_cast` 進行 value conversion。 在最後一種情況下，您必須確保您的物件實際上是子類 instance。
- 使用 `const_cast` 刪除 `const` 限定符（見 const)。
- 使用 `reinterpret_cast` 對 pointer 進行與 int 和其他指標型別的不安全轉換，包括 `void*` 。 僅在您知道自己在做什麼並理解別名問題時才使用此。
- 使用 `bit_cast` 解釋使用相同大小的不同型別的 raw bits，例如將 `double` 的 bits 解釋為 `int64_t` 。(after C++ 17)

有關使用 `dynamic_cast` 的說明，請參閱 RTTI 部分。

34. Streams

> 一律使用 `LOAGCATI/E/W/D` 而非標準的 C++ stream API。

Definition:

Stream 是 C++ 中的 standard I/O abstraction，像是 `<iostream>` 就是例證。 主要用於除錯日誌記錄和測試診斷。

Pros:

`<<` 和 `>>` stream 運算子為格式化的 I/O 提供了一個 API，該 API 易於學習、可移植、可重複使用和可擴充套件。相比之下，printf 甚至不支援std::string，更不用說使用者定義的型別了，而且非常難以移植使用。printf還要求您在該功能的眾多略微不同的版本中進行選擇，並導航幾十個轉換說明符。

Stream 透過 `std::cin` 、 `std::cout` 、 `std::cerr` 和 `std::clog` 為console I/O提供一流的支援。 C API也一樣，但由於需要手動緩衝輸入而受到阻礙。

Cons:

- Stream formatting 可以透過改變 stream 的狀態來配置。 這種改變是永久的，因此您的程式行為可能會受到 stream 之前整個歷史的影響，除非您每次其他程式可能接觸到它時都會竭盡全力將其恢復到已知狀態。 使用者程式不僅可以修改內建狀態，還可以透過註冊系統新增新的狀態變數和行為。
- 由於上述問題、 stream code 和 data 在 streaming code 中混合的方式以及使用運算子 overload（可能會選擇與您預期不同的重載），因此很難精確控制 stream output。
- Stream API 是微妙而複雜的，因此程式設計師必須開發經驗才能有效地使用它。
- 解決 `<<` 的許多過載對編譯器來說是極其昂貴的。 當在大型程式庫中普遍使用時，它可以消耗多達 20% 的解析和語義分析時間。

Decision:

使用 `LOAGCATI/E/W/D` 而非標準的 C++ stream API 來防止上述不確定狀態。

35. Preincrement and Predecrement

> 除非您需要後綴語義，否則請使用增量和遞減運算子的前綴形式（ `++i` ）。

Definition:

當變數遞增（ `++i` 或 `i++` ）或遞減（ `--i` 或 `i--` ）且不使用 value of the expression 時，必須決定是預增量（遞減量）還是後增量（遞減量）。

Pros:

後綴增量/遞減表示式計算為修改前的值。 這可能會導致程式更緊湊但更難閱讀。 前綴形式通常更具可讀性，效率從未降低，並且可以更高效，因為它不需要像後綴那樣先複製值。

Cons:

在 C 中發展出使用後增量的傳統，即使不使用 value of the expression，特別是在迴圈中。

Decision:

使用前綴增量/遞減，除非程式明確需要後綴增量/遞減表達式的結果。

36. Use of const

在 API 中，只要有意義就使用 `const` 。 `constexpr` 是 `const` 的某些用途的更好選擇。

Definition:

宣告的 variables 和 parameters 之前可以加上關鍵字 `const` ，以表示變數沒有更改（例如， `const int foo` ）。 Class 函式可以具有 `const` ，以指示函式不會更改 class 成員變數的狀態（例如， `class Foo { int Bar(char c) const; };` ）。

Pros:

人們更容易理解變數是如何被使用的。 允許編譯器更好地進行型別檢查，並可以想象地生成更好的程式。 幫助人們說服自己相信程式的正確性，因為他們知道他們呼叫的函式在如何修改你的變數方面是有限的。 幫助人們瞭解在多執行緒程式中無需鎖即可安全使用哪些功能。

Cons:

`const` 有一問題：如果您將 `const` 變數傳遞給函式，該函式的原型中必須有 `const` （否則該變數將需要 `const_cast` ）。 在呼叫庫函式時，這可能是一個特殊問題。

Decision:

我們強烈建議在 API 中使用 `const` （即 function parameters, methods, and non-local variables），只要它有意義和準確。 這主要是可讓編譯器驗證的關於操作可否可以改變物件。 擁有一致可靠的方法來區分讀取和寫入，對於編寫執行緒安全程式至關重要，在許多其他上下文中也很有用。 特別是：

- 如果函式保證它不會修改透過引用或指標傳遞的 parameters，相應的函式 parameters 應分別為 `const T&` 或 `const T*` 。

- 對於 pass by value 的函式 parameters， `const` 對呼叫者沒有影響，因此在函式宣告中不建議使用。 參見 TotW #109。
- 除非 methods 改變物件的邏輯狀態（或允許使用者修改該狀態，例如，透過返回non-const reference，但這種情況很罕見），否則無法安全地同時呼叫方法。

在區域性變數上使用 `const` 既不鼓勵也不鼓勵。

Class 的所有 const 操作都應該可以安全地同時呼叫。 如果這不可行，則必須將該類明確記錄為“執行緒不安全”。

Where to put the const:

請將 `const` 放到第一位，如： `const int* foo` 。 把 `const` 放在第一位可以說是更易讀的，因為它跟隨英語將“形容詞”（const）放在“名詞”（int）之前。

37. Use of constexpr, constinit, and consteval

> 使用 `constexpr` 定義真正的常量或確保常量初始化。 使用 `constinit` 來確保非 `const` 變數的 constant initialization。
> 註： `constinit` 和 `consteval` 需至 C++ 20 後採納

Definition:

一些變數可以被宣告為 `constexpr` ，以表明變數是"真。const"，即在編譯/連結時固定。 一些函式和 constructor 可以宣告 `constexpr` ，這使它們能夠用於定義 `constexpr` 變數。 可以將函式宣告為 `consteval` ，以限制其用於編譯時間。

Pros:

使用 `constexpr` 可以定義 floating-point expressions 的常量，而不僅僅是文字；定義 constants of user-defined types；以及 constants with function calls。

Cons:

如果以後必須 de-const，過早地將某物標記為 `constexpr` 可能會導致遷移問題。 目前對 `constexpr` 函式和 constructor 中允許的限制可能會在這些定義中引入難懂的 workarounds。

Decision:

`constexpr` 定義可以對介面的 `const` 部分進行更強大的規範。 使用 `constexpr` 指定真正的常量和支援其定義的函式。 `consteval` 可用於執行時r禁止調用的程式。 避免使函式定義複雜化，以便將其與 `constexpr` 一起使用。 不要使用 `constexpr` 或 `consteval` 來強制 inlining。

38. Integer Types

> 在內建的 C++ 整數型別中，唯一使用的是 `int` 。 如果程式需要不同大小的整數型別，請使用 `<cstdint>` 的精確寬度整數型別，例如 `int16_t` 。 如果您的值可能大於或等於 2^31，請使用64位型別，如 `int64_t` 。 請記住，即使您的 value 對於 int 來說永遠不會太大，它也可能用於可能需要更大型別的中間計算。 如有疑問，請選擇更大的型別。
>
> 儘量避免 unsigned integers。不要使用 unsigned types 來說一個數字永遠不會是負數。 相反，為此使用 assertions。

Definition:

C++ 沒有為 `int` 等整數型別指定確切的大小。 當代架構的常見尺寸: `short` 為 16 bits， `int` 為 32 bits， `long` 為 32 或 64 bits， `long long` 為 64 bits，但不同的平臺會做出不同的選擇，特別是 `long` 。

Pros:

宣告的統一性。

Cons:

C++ 中整數型別的大小可能因編譯器和架構而異。

Decision:

Standard library header `<cstdint>` 定義了 `int16_t` 、 `uint32_t` 、 `int64_t` 等型別。 當您需要保證整數的大小時，您應該優先使用這些，而不是 `short` 、 `unsigned long` 、 `long` 等。 請省略這些型別的 `std::` 前綴，以保持與 `int` 的命名一致性。 在內建的整數型別中，只應使用 `int` 。 適當時，歡迎您使用 `size_t` 和 `ptrdiff_t` 等標準別名。

我們經常使用 `int` ，因為我們知道整數不會太大，例如迴圈計數器。 用普通的舊 `int` 來做這些事情。 您應該假設 `int` 至少是32位，但不要假設它超過32位。 如果您需要64位整數型別，請使用 `int64_t` 或 `uint64_t` 。

對於我們知道可能是“大”的整數，請使用 `int64_t` 。

您不應該使用無符號的整數型別，如 `uint32_t` ，除非有合理的原因，如表示 bit pattern 而不是數字，或者您需要定義 overflow modulo 2^N。 特別是，不要使用 unsigned types 來說一個數字永遠不會是負數。 相反，為此使用 assertions。

如果您的程式是返回大小的容器，請務必使用能夠容納容器任何可能用途的型別。 如有疑問，請使用較大的型別而不是較小的型別。

轉換整數型別時要小心。 不同大小的轉換可能會導致 undefined behavior，導致安全漏洞和其他問題。

On Unsigned Integers:

Unsigned integers 適合表示位 bitfields 和 modular arithmetic。 由於歷史包袱，C++ 標準還使用 unsigned integers 來表示容器的大小 - 標準機構的許多成員認為這是一個錯誤，但目前實際上不可能修復。 事實上，無符號算術無法模擬簡單整數的行為，而是由標準定義，以模擬 modular arithmetic（implicit conversion 所造成的 overflow/underflow），這意味著編譯器無法診斷一類重大的錯誤。 在其他情況下，定義的行為會阻礙最佳化。

白話的說明，考慮以下例子：

```
int a = -1;
unsigned int b = 1;

if (a < b)
     printf("a < b\n");
else
     printf("a > b\n"); // The result!
```

結果很驚人，-1 居然比 1 還大。這是由於 C 語言在比較他們大小時會進行 implicit conversion。如果執行的是 `if ((unsigned int)a < b)` 則 -1 被轉換成 4294967295，結果是 a > b；如果執行的是 `if (a < (int)b)` 則結果是 a < b。採取哪種方式依賴於編譯器。

也就是說，混合整數型別的符號性會導致問題。 我們能提供的最佳建議是：嘗試使用iterator 和容器，而不是 pointer 和 `size()`，儘量不要混合 signed/unsigned，並儘量避免 unsigned types（表示 bitfields 或 modular arithmetic 除外）。 不要僅僅為了斷言變數是非負的而使用 unsigned types。

39. 64-bit Portability

40. Preprocessor Macros

> 避免定義 macros，特別是在 headers 中；請優先使用 inline functions、enums、和 const variables。 使用專案名稱當前綴來命名 macros。 不要使用 macros 來定義 C++ API 的片段。

macros 意味著您看到的程式與編譯器看到的程式不同。 這可能會帶來意想不到的行為，特別是因為 macros 具有全域性範圍。

macros 引入的問題在用於定義 C++ API 的片段時尤為嚴重，而對於 public API 來說更是如此。當開發人員錯誤使用該介面時，編譯器發出的每條錯誤訊息現在都必須解釋 macros 如何形成介面。 重構和分析工具在更新介面時變得很困難。 因此，我們禁止以這種方式使用 macros。 例如，避免以下模式：

```
class WOMBAT_TYPE(Foo) {
  // ...

 public:
  EXPAND_PUBLIC_WOMBAT_API(Foo)

  EXPAND_WOMBAT_COMPARISONS(Foo, ==, <)
};
```

幸運的是，macros 在 C++ 中不像在 C 中那麼必要。 與其使用 macros 來 inline 效能敏感的程式，不如使用 inline functions。 與其使用 macros 來儲存常量，不如使用 const variables。 與其使用 macros 來“縮寫”長變數名稱，不如使用 `using` 語法。 與其使用巨集有條件地編譯程式.....好吧，完全不要這樣做（當然，除了 `#define` guards，以防止 headers 的雙重 include）。 這使測試變得更加困難。

Macros 可以做一些其他技術做不到的事情，特別常常是在 lower-level libraries 中看到。 他們的一些特殊功能（如 stringifying、concatenation 等）無法透過語言本身獲得。 但在使用 macro 之前，請仔細考慮是否有 non-macro 方法來實現相同的結果。 如果您需要使用 macro 來定義介面，請聯絡您的專案主管以請求放棄此規則。

以下使用方式將避免 macros 的許多問題；如果您使用 macros，請儘可能遵循它：

- 不要在.h檔案中定義巨集。
- `#define` macro 的位置請寫在你使用它們之前，然後 `#undef` 它們。
- 在用您自己的 macros 替換之前，不要只是 `#undef` 一個現有的 macros；相反，選擇一個可能唯一的名字。
- 儘量使每個 macros 相互獨立，或者至少很好地記錄該行為。
- 不要使用 `##` 來生成函式/類/變數名稱。

強烈不鼓勵從 headers 匯出 macros（即在 headers 中定義 macros 而不在 headers 結束前 #undefing 它們）。 如果您確實從 headers 匯出 macros，它必須具有全域性唯一的名稱。 要實現這一點，必須使用由專案 namespace 名稱（但大寫）組成的前綴來命名。

41. 0 and nullptr/NULL

> 對 pointer 使用 `nullptr` ，對 chars 使用 `“\0”` 。

對於指標（地址值），請使用 `nullptr` ，因為這提供了型別安全。

對空字元使用 `“\0”` 。 使用正確的型別使程式更易讀。

42. sizeof

> 偏好使用 `sizeof(varname)` 而不是 `sizeof(型別)` 。

當您獲取特定變數的大小時，請使用 `sizeof(varname)` 。如果有人現在或以後更改變數型別， `sizeof(varname)` 將適當更新，而不需修改程式碼。 您可以將 `sizeof(型別)` 用於與任何特定變數無關的程式，例如管理外部或內部資料格式的程式，其中使用 `sizeof(varname)` 不方便。

```
MyStruct data;
memset(&data, 0, sizeof(data));


memset(&data, 0, sizeof(MyStruct)); // Bad


if (raw_size < sizeof(int)) {
  LOG(ERROR) << "compressed record not big enough for count: " << raw_size;
  return false;
}
```

43. Type Deduction (including auto)

> 僅當使程式對不熟悉專案的讀者來說更清晰，或者使程式更安全時，才使用 type deduction。 不要僅僅為了避免寫出過長的顯式型別而使用它。
>
> 註：建議閱讀 "Effective Modern C++" 中 Item1~4，有對 type detuction 做詳細說明。

Definition:

以下幾種情境中，C++ 允許（甚至要求）編譯器推導型別，而不用在程式中完整寫出：

- Function template argument deduction
  可以在沒有 explicit template arguments 的情況下呼叫函式模板。編譯器從 function arguments 的型別中推導：

  ```
  template <typename T>
  void f(T t);

  f(0);  // Invokes f<int>(0)
  ```

- auto variable declarations
  變數宣告可以使用 `auto` 關鍵字代替型別。 編譯器從變數的 initializer 中推斷型別，遵循與

function template argument deduction 一樣的推導規則（使用大括號除外，會被推導成 `std::initializer_list<T>`）。

```
auto a = 42;  // a is an int
auto& b = a;  // b is an int&
auto c = b;   // c is an int
auto d{42};   // d is a std::initializer_list<int>
```

`auto` 可以用 `const` 進行限定，可以作為 pointer 或 reference type 的一部分使用，但不能用作 template argument。這種語法的一個罕見變體使用 `decltype(auto)` 而不是 `auto`，在這種情況下，推導的型別是將 `decltype` 應用於 initializer 的結果。

- Function return type deduction

  `auto`（和 `decltype(auto)`）也可以代替函式的 return type (C++ 14 之後)。編譯器從函式主體的返回語句中推斷出 return type，遵循與變數宣告相同的規則：

```
auto f() { return 0; }  // The return type of f is int
```

  Lambda expressions 返回型別可以以同樣的方式推斷，但背後的機制是 template argument deduction。

- Generic lambdas

  Lambda expressions 可以使用 `auto` 關鍵字來代替其一個或多個引數型別。 這導致 lambda 的 call operator 是一個函式模板，而不是一個普通函式，每個 auto function parameter 都有一個單獨的 template parameter：

```
// Sort `vec` in decreasing order
std::sort(vec.begin(), vec.end(), [](auto lhs, auto rhs) { return lhs > rhs; });
```

- Lambda init captures

  Lambda captures 可以有 explicit initializers，可用於宣告全新的變數，而不僅僅是捕獲現有的變數：

```
[x = 42, y = "foo"] { ... }  // x is an int, and y is a const char*
```

  此語法不允許指定型別；相反，它是使用 `auto` 推導規則。

- Class template argument deduction

  請見下一點。

- Structured bindings

  當使用 `auto` 宣告 tuple、struct 或 array 時，您可以為單個元素指定名稱，而不是整個物件的名稱 （C++ 17 之後）；這些名稱被稱為 “structured bindings”，整個宣告被稱為 “structured binding declaration”。 此語法無法指定 enclosing object 或單一名稱的型別：

```
auto [iter, success] = my_map.insert({key, value});
if (!success) {
  iter->second = value;
}
```

`auto` 也可以用 `const` 、 `&` 和 `&&` 來限定，但請注意，這些限定符在技術上適用於匿名 tuple/struct/array，而不是單一 binding。 決定繫結型別的規則相當複雜；結果往往不足為奇，像是即使宣告為引用，繫結型別通常也不會是引用（但無論如何，它們通常都會表現得像引用）。

Pros:

- C++ 型別名稱可能很繁瑣，特別是當它們涉及模板或 namespace 時。
- 當 C++ 型別名稱在單個宣告或小程式區域中重複時，重複可能無法幫助可讀性。
- 有時讓型別被推斷更安全，因為這樣可以避免意外複製或型別轉換的可能性。

Cons:

當型別 explicit 時，C++ 程式通常更清晰，特別是當型別推斷依賴於程式遠端部分的資訊時。 在這樣的表達方式中：

```
auto foo = x.add_foo();
auto i = y.Find(key);
```

如果 y 的型別不太為人所知，或者 y 在許多行之前被宣告，result type 可能無法立即看出。

程式設計師必須瞭解型別推理何時會或不會產生 reference type，或者他們是得到的是一個副本。

如果推導型別被用作介面的一部分，那麼程式設計師可能會更改其型別，同時只打算更改其值，導致客戶端需比預期做更多的更改。

Decision:

基本規則是：僅使用 tyep deduction 來使程式更清晰或更安全，不要僅僅為了避免編寫顯式型別的不便而使用它。 在判斷程式是否更清晰時，請記住，您的讀者不一定在您的團隊中，也不一定熟悉您的專案，因此儘管會造成您撰寫上的雜亂，但通常會向他人提供有用的資訊。 例如，您可以假設 `std::make_unique<Foo>()` 的返回型別是顯而易見的，但 `MyWidgetFactory()` 的返回型別可能不是。

這些原則適用於所有形式的 tyep deduction，但細節各不相同，如下方所述：

- **Function template argument deduction:**
  此類型推導可被允許。型別推理是與函式模板互動的預設方式，因為它允許函式模板像無限集合的普通函式 overload 一樣。 因此，函式模板的設計幾乎總是使 function template argument deduction 清晰安全，或者無法編譯。
- **Local variable type deduction:**
  對於區域性變數，請謹慎使用 type deduction，透過消除明顯或不相關的型別資訊來使程式更清晰，以便讀者可以專注於程式的有意義部分：

```
std::unique_ptr<WidgetWithBellsAndWhistles> widget =
  std::make_unique<WidgetWithBellsAndWhistles>(arg1, arg2);
absl::flat_hash_map<std::string,
                    std::unique_ptr<WidgetWithBellsAndWhistles>>::const_iterator
    it = my_map_.find(key);
std::array<int, 6> numbers = {4, 8, 15, 16, 23, 42};


// Better!
auto widget = std::make_unique<WidgetWithBellsAndWhistles>(arg1, arg2);
auto it = my_map_.find(key);
std::array numbers = {4, 8, 15, 16, 23, 42};
```

  型別有時包含有用的資訊，如上面的示例：很明顯， `it` 是一個 iterator，在許多情況中，container 或 key type 都不重要，但 value 可能是有用的。 在這種情況下，通常可以用顯式型別定義區域性變數來傳達清楚的資訊：

```
if (auto it = my_map_.find(key); it != my_map_.end()) {
  WidgetWithBellsAndWhistles& widget = *it->second;
  // Do stuff with `widget`
}
```

  一個簡單的原則是：對於涉及到標準庫的 return type，使用 `auto` 可使程式簡潔；而當涉及自定義函式/類，使用顯式型別定義可傳達更清楚的資訊。
  盡量不要使用 `decltype(auto)` ，因為它是一個相當晦澀的功能，所傳達的資訊相當模糊。
- **Return type deduction:**
  僅當函式主體的 `return` 數量很少，才使用 return type deduction（適用於函式和lambdas），否則讀者可能無法一目瞭然地知道 return type 是什麼。 此外，僅當函式或lambda 是 narrow scope 時才使用它，因為此類型的函式不會定義抽象邊界：implementation 是介面。 特別是，header 檔案中的 public 函式禁止使用 return type deduction。
- **Parameter type deduction:**
  不要使用 lambdas 的 `auto` parameters，因為實際型別由呼叫 lambda 的程式決定，而不是由 lambda 的定義決定。 因此，顯式型別更加的清晰。

- **Lambda init captures:**
  Init captures 被更具體的規則，請參閱。
- **Structured bindings:**
  與其他形式的 type deduction 不同，structured bindings 實際上可以透過為較大物件的元素賦予有意義的名稱來使資訊更清楚。當物件是 pair 或 tuple 時，structured bindings 特別有益（如上面的例子所示），因為它們一開始就沒有有意義的欄位名稱，但請注意，除非像 `insert` 這樣的預先存在的API強迫您使用，否則您通常不應該使用 pair 或 tuple。
  如果被 binding 的物件是一個 struct，不建議使用 structured bindings，因為額外的名稱不會比他本身的命名來的更有意義。

44. Class Template Argument Deduction

> 不要使用 class template argument deduction。

Definition:

Class template argument deduction（通常縮寫為“CTAD”）發生在變數以模板的型別宣告時，並且沒有提供 template argument list（沒有<>括號）：

```
std::array a = {1, 2, 3};  // `a` is a std::array<int, 3>
```

編譯器使用模板的 “deduction guides” 從 initializer 中推斷引數，該 guides 可以是 explicit 或 implicit。

Explicit deduction guides 看起來像帶有 trailing return types 的函式宣告，函式名稱是模板的名稱。 例如，上述示例依賴於 `std::array` 的 explicit deduction guides：

```
namespace std {
template <class T, class... U>
array(T, U...) -> std::array<T, 1 + sizeof...(U)>;
}
```

Primary template 中的 constructor（而不是 template specialization）也隱含地定義了 deduction guides。

當您宣告依賴 CTAD 的變數時，編譯器使用 constructor overload 選擇 deduction guides，該 guides 的 return types 成為變數的型別。

Pros:

CTAD 有時允許您從程式中省略樣板語法。

Cons:

從 constructor 生成的 implicit deduction guides 可能具有不良行為，或者完全不正確。 對於 CTAD 在 C++17 中引入之前編寫的 constructor 來說，這尤其成問題，因為這些 constructor 的作者無法知道（更不用說修復）他們的建構函式會給 CTAD 帶來的任何問題。此外，新增 explicit deduction guides 來修復這些問題可能會破壞任何依賴 implicit deduction guides 的現有程式。

CTAD 還存在許多與 `auto` 相同的缺點，因為它們都是從 initializer 中推導出全部或部分變數型別的機制。 CTAD確實為讀者提供了比 `auto` 更多的資訊，但它也沒有給讀者一個明顯的線索，即資訊被省略了。

Decision:

不要使用 CTAD，因為客戶端無法清楚得知模板是否提供至少一個 explicit deduction guides 來支援使用 CTAD。

45. Designated Initializers

> [TODO] 在升級至 C++ 20 之後再補充此規範。

46. Lambda Expressions

> 斟酌使用 lambda expressions。當 lambda 將退出當前 scope 時，偏好使用 explicit captures。

Definition:

Lambda expressions 是建立匿名函式物件的簡明方式。 在將函式作為引數傳遞時，它們通常很有用。例如：

```
std::sort(v.begin(), v.end(), [](int x, int y) {
  return Weight(x) < Weight(y);
});
```

它們進一步允許從 enclosing scope 中捕獲變數，要麼顯式地透過名稱，要麼使用隱式地捕獲（預設）。 explicit captures 要求將每個變數 value or reference capture：

```
int weight = 3;
int sum = 0;
// Captures `weight` by value and `sum` by reference.
std::for_each(v.begin(), v.end(), [weight, &sum](int x) {
  sum += weight * x;
});
```

預設的隱式捕獲使用 captures by reference，包含 `this` 以及任何被使用的 member variables：

```
const std::vector<int> lookup_table = ...;
std::vector<int> indices = ...;
// Captures `lookup_table` by reference, sorts `indices` by the value
// of the associated element in `lookup_table`.
std::sort(indices.begin(), indices.end(), [&](int a, int b) {
  return lookup_table[a] < lookup_table[b];
});
```

Variable capture 也可以有一個顯式 initializer，可用於 captures move-only variables by value，或用於 ordinary reference 或 value captures的其他情況：

```
std::unique_ptr<Foo> foo = ...;
[foo = std::move(foo)] () {
  ...
}
```

此類捕獲（通常稱為 "init captures" 或 "generalized lambda captures"）實際上不需要從 enclosing scope “捕獲”任何東西，甚至不需要在enclosing scope 中具有名稱；這種語法是定義 lambda 物件成員的完全通用的方式：

```
[foo = std::vector<int>({1, 2, 3})] () {
  ...
}
```

捕獲型別的推導規則與 `auto` 相同。

Pros:

- Lambdas 比定義要傳遞給STL演算法的函式物件的其他方法要簡潔得多，這可以提高可讀性。
- 適當使用預設捕獲可以消除冗餘，並從預設中突出顯示重要異常。

- Lambdas、`std::function` 和 `std::bind` 可以組合用作通用 callback 機制；它們使編寫將 functions 作為 parameters 的函式變得容易。

Cons:

- Lambda 中的變數捕獲可能是 dangling-pointer bugs 的來源，特別是當lambda 離開當前 scope 時。
- Default captures by value 可能具有誤導性，因為它們不能防止 dangling-pointer bugs。 Capturing a pointer by value 不會導致 deep copy，因此它通常具有與 capture by reference 相同的生命週期問題。 當 captures `this` by value 時，這尤其令人困惑，因為使用它通常是implicit 的。
- 捕獲實際上宣告新變數（無論捕獲是否有 initializer），但它們看起來與 C++ 中的任何其他變數宣告語法完全不同。 特別是，變數的型別，甚至 `auto` 都沒有位置（儘管 Init captures 可以間接指向它，例如，使用 cast）。 這甚至很難將它們識別為宣告。
- Init captures本 質上依賴於 type deduction，並存在許多與 `auto` 相同的缺點，還有一個問題，即語法甚至沒有提示讀者正在發生 deduction。
- 使用lambdas可能會失控；非常長的 nested 匿名函式可能會使程式更難理解。

Decision:

- 斟酌使用 lambda expression，格式如下所述。
- 如果 lambda 可能離開當前 scope，則更喜歡顯式捕獲。 例如，不要這樣：

```
{
  Foo foo;
  ...
  executor->Schedule([&] { Frobnicate(foo); })
  ...
}
// BAD! The fact that the lambda makes use of a reference to `foo` and
// possibly `this` (if `Frobnicate` is a member function) may not be
// apparent on a cursory inspection. If the lambda is invoked after
// the function returns, that would be bad, because both `foo`
// and the enclosing object could have been destroyed.
```

偏好使用：

```
{
  Foo foo;
  ...
  executor->Schedule([&foo] { Frobnicate(foo); })
  ...
}
// BETTER - The compile will fail if `Frobnicate` is a member
// function, and it's clearer that `foo` is dangerously captured by
// reference.
```

- 僅當 lambda 的生命週期明顯短於任何潛在捕獲時，才使用預設引用捕獲（ `[&]` ）。
- 僅使用 default capture by value（ `[=]` ）作為為短 short lambda 捕獲變數的手段，其中捕獲的變數集一目瞭然，這不會導致隱式捕獲。（這意味著出現在 non-static class member function 的 lambda 會捕獲 `this` 或透過[&] 捕獲。） 更喜歡不寫長或複雜的帶有default capture by value 的 lambdas。
- 僅使用捕獲來從 enclosing scope 中捕獲變數。 不要使用帶有 initializer 的捕獲來引入新名稱，或大幅改變現有名稱的含義。 相反，以常規方式宣告一個新的變數，然後捕獲它。
- 有關指定 parameters 和 return types 的指引，請參閱 type deduction。

47. Template Metaprogramming

> 避免複雜的 template programming

Definition:

Template metaprogramming 是指利用 C++ 模板例機制完成的一系列技術，可用於在 type domain 中執行任意 compile-time 計算。

Pros:

Template metaprogramming 允許 type safe 和高效能且靈活的介面。 沒有它，像 GoogleTest、 `std::tuple` 、 `std::function` 和 `Boost.Spirit` 是不可能實作的。

Cons:

除了 C++ 高手，template metaprogramming 中使用的技術往往對任何人來說都很模糊。以複雜方式使用模板的程式通常無法讀取，並且很難除錯或維護。

Template metaprogramming 通常會導致極慢的編譯時間和不良的錯誤訊息：即使介面很簡單，當使用者做錯事時，複雜的實現細節也會變得可見。

Decision:

Template metaprogramming 有時能達到更乾淨、更易於使用的介面，但過於聰明也往往是一種誘惑。它最好用於少數低級別元件，這些元件的額外維護負擔分散在大量用途上。

在使用 template metaprogramming 或其他複雜的模板技術之前，請三思而後行；考慮您的團隊中的普通成員是否能夠在您切換到另一個專案後充分理解您的程式來維護它，或者非C++程式設計師或隨意瀏覽程式庫的人是否能夠理解錯誤訊息或跟蹤他們想要呼叫的函式的流程。 如果您正在使用 recursive template instantiations、type lists、 metafunctions、expression templates，或依靠 SFINAE 或 `sizeof` 來檢測 function overload resolution，那麼您很有可能走得太遠了。

如果您使用 template metaprogramming ，您應該投入大量精力來最大限度地減少和隔離複雜性。 您應該儘可能隱藏 metaprogramming 作為實現細節，以便面向使用者的 header 具有可讀性，並且您應該確保棘手的程式註釋得特別好。 您應該仔細記錄如何使用程式，並說一些關於“生成”程式的樣子。 當使用者出錯時，請格外注意編譯器發出的錯誤訊息。 錯誤訊息是使用者介面的一部分，您的程式應根據需要進行調整，以便從使用者的角度來看，錯誤訊息是可以理解和操作的。

48. Concepts and Constraints

> [TODO] 在升級至 C++ 20 之後再補充此規範。

49. Boost

Definition:

Boost library collection 是一個流行的、免費、開源的C++庫集合。

Pros:

Boost 程式通常品質很高，可以容易嵌入，並填補了 C++ 標準庫中的許多重要空白，如 type traits 和更好的 `binders` 。

Cons:

一些 Boost 庫可能妨礙可讀性的編碼實踐，如 metaprogramming 和其他高階模板技術，以及過度“功能性”的寫程式風格。

Decision:

為了保持高水準的可讀性，我們只允許已批准的 Boost功能子集。 目前，以下 libraries 是允許的：

- Call Traits from `boost/call_traits.hpp`
- Compressed Pair from `boost/compressed_pair.hpp`

- The Boost Graph Library (BGL) from `boost/graph`, except serialization (`adj_list_serialize.hpp`) and parallel/distributed algorithms and data structures (`boost/graph/parallel/*` and `boost/graph/distributed/*`).
- Property Map from `boost/property_map`, except parallel/distributed property maps (`boost/property_map/parallel/*`).
- Iterator from `boost/iterator`
- The part of Polygon that deals with Voronoi diagram construction and doesn't depend on the rest of Polygon: `boost/polygon/voronoi_builder.hpp`, `boost/polygon/voronoi_diagram.hpp`, and `boost/polygon/voronoi_geometry_type.hpp`
- Bimap from `boost/bimap`
- Statistical Distributions and Functions from `boost/math/distributions`
- Special Functions from `boost/math/special_functions`
- Root Finding & Minimization Functions from `boost/math/tools`
- Multi-index from `boost/multi_index`
- Heap from `boost/heap`
- The flat containers from Container: `boost/container/flat_map`, and `boost/container/flat_set`
- Intrusive from `boost/intrusive`.
- The boost/sort library.
- Preprocessor from `boost/preprocessor`.

此列表可能會在未來擴充。

50. Other C++ Features

與 Boost 一樣，一些現代 C++ 擴充套件鼓勵妨礙可讀性的編碼實踐 - 例如，透過刪除可能對讀者有幫助的額外檢查（如型別名稱），或鼓勵 template metaprogramming。 其他擴充套件重複了現有機制提供的功能，這可能會導致混淆和轉換成本。

Decision:

除了樣式指南其餘部分中描述的內容外，以下C++功能可能無法使用：

- Compile-time rational numbers（`<ratio>`），因為它更加的靠近 template-heavy interface style。
- `<cfenv>` 和 `<fenv.h>` headers，因為許多編譯器不支援這些功能。
- < `filesystem` > header 沒有足夠的測試支援，並且存在固有的安全漏洞。

51. Nonstandard Extensions

> 適當使用 C++ 的非標準擴充套件。

Definition:

編譯器支援不屬於標準 C++ 的各種擴充套件。 此類擴充套件包括 GCC 的 `__attribute__` ，intrinsic functions 如 `__builtin_prefetch` 或SIMD， `#pragma` ，inline assembly， `__COUNTER__` ， `__PRETTY_FUNCTION__` ，複合語句表示式（例如， `foo = ({ int x; Bar (&x) ; x})` ，可變長度陣列和 `alloca()` ，以及“Elvis運算子” `a ? : b` 。

Pros:

- 非標準擴充套件可能會提供標準C++中不存在的有用功能。
- 編譯器的重要效能指導只能使用擴充套件來指定。

Cons:

- 非標準擴充套件並不適用於所有編譯器。 使用非標準擴充套件會降低程式的可移植性。
- 即使所有目標編譯器都支援它們，但擴充套件通常沒有明確指定，編譯器之間可能存在微妙的行為差異。
- 非標準擴充套件增加了讀者必須知道才能理解程式的語言功能。
- 非標準擴充套件需要額外的工作來跨架構移植。

Decision:

雖然使用非標準擴充套件會降低程式的可移值性，但這種情況在不公開的專案中並不常見。考慮方便性，我們允許合理的使用它。

52. Aliases

> 儘量不使用 public aliases，若 public aliases 是為了 API 使用者的方便，應該有明確的說明。請優先使用 `using` 而非 `typedef` 。

Definition:

有幾種方法可以建立 aliases：

```
using Bar = Foo;
typedef Foo Bar;  // But prefer `using` in C++ code.
using ::other_namespace::Foo;
using enum MyEnumType;  // Creates aliases for all enumerators in MyEnumType.
```

在新程式中， `using` 比 `typedef` 更可取，因為它提供了與 C++ 其餘部分更一致的語法，並與模板配合使用。

與其他宣告一樣，在 header 中宣告的 aliases 是該 header public API 的一部分，除非它們在函式定義中，在類的私有部分中，或者在顯式標記的內部命名空間中。 這些區域或.cc檔案中的別名是實作細節（因為客戶端程式不能 include 它們），不受此規則的限制。

Pros:

- Aliases 可以透過簡化一個長或複雜的名稱來提高可讀性。
- Aliases 可以透過在一個地方命名API中重複使用的型別來減少重複，這可能會使以後更容易更改型別。

Cons:

- 當放置在客戶端程式可以 include 的 headers 中時，aliases 會增加該header API 中的實體數量，從而增加其複雜性。
- 客戶可以很容易地依賴 public aliases，這使得更改變得困難。
- 建立一個僅用於 implementation 的 public aliases 可能很誘人，但會忽略其對 API 或可維護性的影響。
- Aliases 可能會造成名稱碰撞的風險。
- Aliases 可以透過給熟悉的 constructor 一個陌生的名字，降低可讀性
- Type aliases 容易建立一個不明確的 API 約定：不清楚該 aliases 是否保證與其 aliases 的型別相同，具有相同的 API，還是只能以特定的方式使用。

Decision:

不要在 public API 中放置 aliases；只有當您希望客戶使用它時，才這樣做。

在定義 public aliases 時，請說明新名稱的意圖，包括它是否保證始終與當前 aliases 的型別相同，或者是否打算提供更有限的相容性。 這讓使用者知道他們是否可以將型別視為可替代的，或者是否必須遵循更具體的規則，並可以幫助實現保留一定程度的自由來更改別名。

不要在 public API 中放置 namespace aliases。（另見 Namespaces）。

例如，這些 aliases 記錄了如何在客戶端程式中使用它們：

```
namespace mynamespace {
// Used to store field measurements. DataPoint may change from Bar* to some internal ty
// Client code should treat it as an opaque pointer.
using DataPoint = ::foo::Bar*;

// A set of measurements. Just an alias for user convenience.
using TimeSeries = std::unordered_set<DataPoint, std::hash<DataPoint>, DataPointCompara
}  // namespace mynamespace
```

這些別名不記錄預期用途，其中一半不供客戶端使用：

```
namespace mynamespace {
// Bad: none of these say how they should be used.
using DataPoint = ::foo::Bar*;
using ::std::unordered_set;  // Bad: just for local convenience
using ::std::hash;           // Bad: just for local convenience
typedef unordered_set<DataPoint, hash<DataPoint>, DataPointComparator> TimeSeries;
}  // namespace mynamespace
```

然而，local convenience aliases 在函式定義、類的 private 部分、顯式標記的 internal namespaces 和.cc檔案中是適當的：

```
// In a .cc file
using ::foo::Bar;
```

53. Switch Statements

> 如果不是以 enumerated value 為條件， `switch` 語句應始終具有 `default` case（如果是 enumerated value，如果未處理任何值，編譯器將警告您）。 如果預設情況永遠不應該執行，請將其視為錯誤。

```
switch (var) {
  case 0: {
    ...
    break;
  }
  case 1: {
    ...
    break;
  }
  default: {
    LOG(FATAL) << "Invalid value in switch statement: " << var;
  }
}
```

在 C++ 17 之後，從一個 case 到 case 必需加上 `[[fallthrough]]` 。 `[[fallthrough]]` 應放在下一個 case 執行點之前。 一個常見的例外是連續的 case 且中間沒有執行程式，在這種情況下不需要加上此屬性標籤。

```
switch (x) {
  case 41:  // No annotation needed here.
  case 43:
    if (dont_be_picky) {
      // Use this instead of or along with annotations in comments.
      [[fallthrough]];
    } else {
      CloseButNoCigar();
      break;
    }
  case 42:
    DoSomethingSpecial();
    [[fallthrough]];
  default:
    DoSomethingGeneric();
    break;
}
```

Naming

54. General Naming Rules

使用即使對不同團隊的人來說也很清楚的名字來最佳化可讀性。

使用描述物件目的或意圖的名稱。 不要擔心名稱長短，因為讓新讀者立即理解您的程式要重要得多。 儘量減少使用專案以外的人可能不知道的縮寫（特別是首字母縮寫）。 不要透過刪除單詞中的字母來縮寫。 根據經驗，如果縮寫被列在維基百科中，它可能是可以的。 一般來說，描述性應該與名稱的可見度範圍成正比。 例如， `n` 在只有5行函式中可能是一個很好的變數名稱，但在 class 的範圍內，它可能太模糊了。

```
// Good!!
class MyClass {
 public:
  int CountFooErrors(const std::vector<Foo>& foos) {
    int n = 0;  // Clear meaning given limited scope and context
    for (const auto& foo : foos) {
      ...
      ++n;
    }
    return n;
  }
  void DoSomethingImportant() {
    std::string fqdn = ...;  // Well-known abbreviation for Fully Qualified Domain Name
  }
 private:
  const int kMaxAllowedConnections = ...;  // Clear meaning within context
};
```

```
// Bad!!
class MyClass {
 public:
  int CountFooErrors(const std::vector<Foo>& foos) {
    int total_number_of_foo_errors = 0;  // Overly verbose given limited scope and cont
    for (int foo_index = 0; foo_index < foos.size(); ++foo_index) {  // Use idiomatic `
      ...
      ++total_number_of_foo_errors;
    }
    return total_number_of_foo_errors;
  }
  void DoSomethingImportant() {
    int cstmr_id = ...;  // Deletes internal letters
  }
 private:
  const int kNum = ...;  // Unclear meaning within broad scope
};
```

請注意，某些眾所周知的縮寫是可以的，例如 `i` 表示迭代變數， `T` 表示模板引數。

就以下命名規則而言，“單詞” 是指您在沒有內部空格的情況下用英語書寫的任何內容。 這包括縮寫，如首字母縮寫。 對於以混合大小寫（有時也稱為“camel case”或“Pascal case”）書寫的名稱，其中每個單詞的第一個字母都大寫，請偏好將縮寫的第一個字母寫成大寫，例如 `StartRpc()` 而不是 `StartRPC()` 。

Template parameters 應遵循其類別的命名樣式：type template parameters 應遵循型別名稱的規則，non-type template parameters 應遵循變數名稱的規則。

55. File and Folder Names

檔名與資料夾名應全部小寫，名稱超過兩個以上的單詞可以包括下劃線（ `_` ）。

可接受檔名範例：

- `my_useful_class.cc`
- `my_useful_class_test.cc // _unittest and _regtest are deprecated.`
- `/src`
- `/include`
- `/bin`

C++ 檔案應以 .cc 結尾，header 檔案應以 .h 結尾。 依賴於在特定點以文字方式包含的檔案應以.inc結尾（另請參見關於自包含標題的部分）。

不要使用 `/usr/include` 中已經存在的檔名，例如 `db.h` 。

一般來說，讓您的檔名非常具體。 例如，使用 `http_server_logs.h` 而不是 `logs.h` 。 一個常見的情況是，有一對名為 `foo_bar.h` 和 `foo_bar.cc` 的檔案，定義一個名為 `FooBar` 的 class。

56. Type Names

Type names 以大寫字母開頭，每個新單詞都有一個大寫字母，沒有下劃線:
`MyExcitingClass` ， `MyExcitingEnum` 。

所有型別的名稱 - class、struct、type aliases、enums和 type template parameters - 具有相同的命名約定。 型別名稱應以大寫字母開頭，每個新單詞都應有一個大寫字母。 沒有下劃線。 例如：

```
// classes and structs
class UrlTable { ...
class UrlTableTester { ...
struct UrlTableProperties { ...

// typedefs
typedef hash_map<UrlTableProperties *, std::string> PropertiesMap;

// using aliases
using PropertiesMap = hash_map<UrlTableProperties *, std::string>;

// enums
enum class UrlTableError { ...
```

57. Variable Names

變數（包括 function parameters）和 data members 的名稱應為 snake_case（全部小寫，單詞之間有下劃線）。 `class` 的 data members（但不是 `struct` ）還必須加上下劃線結尾。 例如： `a_local_variable` 、 `a_struct_data_member` 、 `a_class_data_member_` 。

若變數表達數學含義（如：矩陣），則另有特殊規範如下方所述。

Common Variable names

```
std::string table_name;  // OK - snake_case.


std::string tableName;   // Bad - mixed case.
```

Class Data Members

Class data members，包括靜態和非靜態的，都像一般變數一樣被命名，但需加上結尾下劃線。

```
class TableInfo {
  ...
 private:
  std::string table_name_;  // OK - underscore at end.
  static Pool<TableInfo>* pool_;  // OK.
};
```

Struct Data Members

`struct` 的 data members，包括靜態和非靜態的，都像普通非成員變數一樣命名。 他們沒有結尾下劃線。

```
struct UrlTableProperties {
  std::string name;
  int num_entries;
  static Pool<UrlTableProperties>* pool;
};
```

Mathmatical Notation

SLAM 的應用中存在大量的線性代數表達。為了貼近我們所使用的數學符號以增進閱讀效率，關於此類變數命名需予以特殊規範。

Matrix

使用大寫字母，並且遵循 row-first 規則。由於對於同一個矩陣，現行存在不同 liberary 的表達方式，因此我們以不同符號作為區分：

- `T` : Transform Matrix，如： `Eigen::Matrix4f` / `Eigen::Matrix4d` 。
- `cvT` : Open CV based Transform Matrix，如：
  `cv::Mat(4, 4, CV_32FC1, data)` / `cv::Mat(4, 4, CV_64FC1, data)` 。
- `R` : Roataion Matrix，如： `Eigen::Matrix3f` / `Eigen::Matrix3d` 。
- `cvR` : Open CV based Roataion Matrix，如：
  `cv::Mat(3, 3, CV_32FC1, data)` / `cv::Mat(3, 3, CV_64FC1, data)` 。
- `SE3` / `SE2`：特殊歐氏群 $\mathrm{SE}(3)$，如： `Sophus::SE3f` / `Sophus::SE3d` 。 `SE2` 以此類推。
- `SO3` / `SO2`：特殊正交群 $\mathrm{SO}(3)$，如： `Sophus::SO3f` / `Sophus::SO3d` 。 `SO2` 以此類推。

Vector

- `t`：Translation vector，如： `Eigen::Vector3f` / `Eigen::Vector3d` 。
- `q`：Quaternion，如： `Eigen::Quaterniond` 。
- `se3` ， `se2`：李代數 $\mathfrak{se3}$，如： `Eigen::Vector6f` / `Eigen::Vector6d` 。
- `so3` ， `so2`：李代數 $\mathfrak{so3}$，如： `Eigen::Vector3f` / `Eigen::Vector3d` 。

Coordinate System

而對於座標系描述我們使用小寫，我們給予以下規範：

- `w`：首幀 cam0 的座標系，設為 cv world coordinate。
- `g`：首幀 cam0 擺正後的座標系，設為 cv gravity coordinate。
- `{i}{n}`：指定 i 幀第 n 個 cam 的座標系。i 可自行定義代號，若代號過於晦澀請加上說明。
- `iw`：首幀 IMU 的座標系，設為 imu world coordinate。
- `ig`：首幀 IMU 對齊重力方向後的座標系，設為 imu gravity coordinate。
- `i{n}`：指定 n 幀 IMU 的座標系。n 可自行定義代號，若代號過於晦澀請加上註解說明。
- 自定義：無法上述規定無法滿足需求，我們允許自定義座標系名稱，且必須在 declaretion 處寫下註解說明。

Linear Algebra

有了上述的保留字，我們便可以定義線性代數運算的符號表達，通用規則為：

```
{Vector}_{coord2} = {Matrix}_{coord2}_{coord1} * {Vector}_{coord1};

{Matrix}_{coord3}_{coord1} = {Matrix}_{coord3}_{coord2} * {Matrix}_{coord2}_{coord1}
```

等效於以下數學表達：

$$\mathrm{t}_{\mathrm{coord.2}} = \mathrm{T}^{\mathrm{coord.1}}_{\mathrm{coord.2}} * \mathrm{t}_{\mathrm{coord.1}}$$

$$\mathrm{T}^{\mathrm{coord.1}}_{\mathrm{coord.3}} = \mathrm{T}^{\mathrm{coord.2}}_{\mathrm{coord.3}} * \mathrm{T}^{\mathrm{coord.1}}_{\mathrm{coord.2}}$$

我們可以更進一步的參考以下範例來看如何命名：

- $\mathrm{t}_{\mathrm{g}} = \mathrm{T}^{\mathrm{w}}_{\mathrm{g}} * \mathrm{t}_{\mathrm{w}}$： `t_g = T_g_w * t_w`
- $\mathrm{T}^{\mathrm{w}}_{\mathrm{cur2}} = \mathrm{T}^{\mathrm{cur0}}_{\mathrm{cur2}} * \mathrm{T}^{\mathrm{w}}_{\mathrm{cur0}}$： `T_cur2_cw = T_cur2_cur0 * T_cur0_w` ， `cur0` 代表當前幀 cam0 座標系。
- $\mathrm{T}^{\mathrm{anchor}}_{\mathrm{ref0}} = \mathrm{T}^{\mathrm{w}}_{\mathrm{ref0}} * \mathrm{T}^{\mathrm{anchor}}_{\mathrm{w}}$： `T_ref0_anchor = T_ref0_w * T_w_anchor` ，其中 `anchor` 為自定義座標系， `ref0` 代表自訂義 `ref` 幀的 cam0 坐標系。

58. Constant Names

被宣告為 `constexpr` 或 `const` 的變數，其值在運行期間是固定的，以前綴“k”命名，後面混合大小寫。 在不能將大寫用於分離的極少數情況下，下劃線可以用作分隔符。 例如：

```
const int kDaysInAWeek = 7;
const int kAndroid8_0_0 = 24;  // Android 8.0.0
```

所有具有 static storage duration 的變數（即靜態和全域性變數，詳情請參閱 storage duration）都應以這種方式命名，這也包括模板的不同 instantiations 可能具有不同值的變數。 對於其他儲存類的變數，例如 local variables，此約定是可選的；否則適用一般的變數命名規則。 例如：

```
void ComputeFoo(std::string_view suffix) {
  // Either of these is acceptable.
  const std::string_view kPrefix = "prefix";
  const std::string_view prefix = "prefix";
  ...
}
```

```
void ComputeFoo(std::string_view suffix) {
  // Bad - different invocations of ComputeFoo give kCombined different values.
  const std::string_view kCombined{kPrefix, suffix};
  ...
}
```

59. Function Names

函式命名應混合大小寫；getter 和 setter 類型則可以像變數一樣命名。

通常，函式應該以大寫字母開頭，每個新單詞都有一個大寫字母。

```
AddTableEntry()
DeleteUrl()
OpenFileOrDie()
```

Get 和 set functions 可以像變數一樣命名。 這些通常對應於實際成員變數，但這不是必需的。 例如， `int count()` 和 `void set_count(int count)` 。

60. Namespace Names

Namespace 名稱都是小寫的，單詞用下劃線分隔。 Top-level namespace 名稱基於專案名稱。 避免 nested namespaces 和眾所周知的 top-level namespaces 之間的衝突。

Top-level namespace 的名稱應該是專案名稱。 該 namescape 中的程式通常應位於其子目錄中。

請記住，禁止縮寫名稱的規則 一樣適用於 namespace。 Namespace 內的程式很少需要提及 namespace 名稱，因此通常不需要縮寫。

避免 nested namespaces 與 top-level namespaces 同名。 由於名稱查詢規則，名稱空間之間的衝突可能會導致 build breaks。 特別是，不要建立任何 nested 的std namespace。 請偏好使用 unique project identifiers （ `websearch::index` ， `websearch::index_util` ），而不是像 `websearch::util` 這樣的容易發生碰撞的名稱。 還要避免過於深的 nested namespaces（TotW #130）。

對於 internal namespaces，請小心將其他程式新增到相同的 internal namespaces 中可能導致衝突。 在這種情況下，使用檔名製作唯一的內部名稱是有幫助的（ `websearch::index::frobber_internal` 在 `frobber.h` 中使用）。

61. Enumerator Names

Enumerators（適用於 scoped 和 unscoped enums）應該像 constants 一樣命名，而不是像 macros 那樣命名。 也就是說，使用 `kEnumName` 而不是 `ENUM_NAME` 。

```
enum class UrlTableError {
  kOk = 0,
  kOutOfMemory,
  kMalformedInput,
};


// Bad!
enum class AlternateUrlTableError {
  OK = 0,
  OUT_OF_MEMORY = 1,
  MALFORMED_INPUT = 2,
};
```

62. Macro Names

通常不會去定義 macro，如果你這樣做了，請這樣命名： `MY_MACRO_THAT_SCARES_SMALL_CHILDREN_AND_ADULTS_ALIKE` 。

請參閱 macros 的描述；一般來說，不應使用 macros。 然而，如果絕對需要它們，那麼它們應該用所有大寫字母和下劃線來命名，並帶有專案的前綴。

```
#define MYPROJECT_ROUND(x) ...
```

63. Exceptions to Naming Rules

如果您要命名的東西類似於現有 C 或 C++ 所定義的東西，那麼您可以遵循現有的命名約定方案。

`bigopen()`：
function name, follows form of open()
`uint`：
typedef
`bigpos`：
struct or class, follows form of pos
`sparse_hash_map`：
STL-like entity; follows STL naming conventions
`LONGLONG_MAX`：
a constant, as in INT_MAX

Comments

Comments 對於保持我們的程式可讀性至關重要。 以下規則描述了您應該 comment 的內容和位置。 但請記住：**雖然 comments 非常重要，但最好的程式是 self-documenting。 為型別和變數提供合理的名稱比使用晦澀的名字要好得多，無法傳達足夠資訊才透過 comments 來解釋。**

64. Comment Style

可任意使用 `//` 或 `/* */` 語法，但請保持前後文風格一致。

65. File Comments

在每個檔案開頭加上 license。

如果 source file（如.h檔案）宣告了多個開放給使用者的 public API (代表客戶是不知道實作的)，請附上描述這些抽象集合的評論。 包括足夠的細節，以便未來的作者知道此檔案的意圖。

例如，如果您為 `frobber.h` 編寫檔案註釋，則無需在 `frobber.cc` 或 `frobber_test.cc` 中包含檔案註釋。 另一方面，如果 `register_objects.cc` 中沒有相關的 header file，則必須在 `register_objects.cc` 中包含檔案註釋。

66. Struct and Class Comments

每個複雜的 class 或 struct 宣告都應該有一個隨附的註釋，描述它的用途以及應該如何使用它。

```
// Iterates over the contents of a GargantuanTable.
// Example:
//    std::unique_ptr<GargantuanTableIterator> iter = table->NewIterator();
//    for (iter->Seek("foo"); !iter->done(); iter->Next()) {
//      process(iter->key(), iter->value());
//    }
class GargantuanTableIterator {
  ...
};
```

67. Function Comments

宣告註釋描述函式的使用（當它晦澀時）；函式定義的註釋描述操作。

Function Declarations

幾乎每個函式宣告都應該在它之前有註釋，描述函式的作用以及如何使用它。 只有當函式簡單而明顯時，才能省略這些註釋（例如，setters）。 在.cc檔案中宣告的 private methods 和函式不能豁免。 函式註釋盡量以動詞作為開頭；例如，"Opens the file"。 一般來說，這些註釋並不描述該函式如何執行其任務。 相反，這應該留給 function definition 中的註釋。

在 function declaration 的 comments 中通常包含：

- Inputs 和 output 是什麼。
- 對於 class member functions：物件是否在方法 duration of the method call 之後記住 reference 或 pointer parameters。 這常於 constructor 的pointer/reference parameters。

- 對於每個 pointer parameters，是否允許為 null，如果是 null，會發生什麼。
- 對於每個 input/output argument，該引數處於的任何狀態會發生什麼。 （例如，狀態是附加的還是覆蓋的？）
- 如果特定的使用函式會造成任何效能影響。

這裡有一個例子：

```
// Returns an iterator for this table, positioned at the first entry
// lexically greater than or equal to `start_word`. If there is no
// such entry, returns a null pointer. The client must not use the
// iterator after the underlying GargantuanTable has been destroyed.
//
// This method is equivalent to:
//    std::unique_ptr<Iterator> iter = table->NewIterator();
//    iter->Seek(start_word);
//    return iter;
std::unique_ptr<Iterator> GetIterator(absl::string_view start_word) const;
```

然而，不要註解完全明顯的事實。

當記錄 function overrides 時，請關注重寫本身的細節，而不是重複重寫函式的註釋。 在其中許多情況下，覆蓋不需要額外的說明，因此不需要 comment。

在 comment constructor 和 destructor 時，請記住，閱讀程式的人知道 constructor 和 destructor 的用途，所以只說 “destroys this object”之類的註釋是沒有用的。 記錄 constructor 如何處理他們的 parameters（例如，如果他們擁有 pointers 的所有權），以及 destructor 做了什麼清理。 如果這微不足道，只需跳過評論。 destructor 沒有註釋是很常見的。

Function Definitions

如果函式複雜，function definition 應該有一個解釋性的評論。 例如，在定義註釋中，您可以描述您使用的任何技巧，概述您所經歷的步驟，或解釋您為什麼選擇以自己的方式實現該函式。 例如，您可能會提到為什麼它必須為功能的前半部分使用 `lock` ，但為什麼後半部分不需要它。

請注意，您不應該給出在.h檔案或任何地方重複的註釋。 簡要總結一下該功能的作用是可以的，但評論的重點應該是它是如何做到的。

68. Variable Comments

一般來說，變數的實際名稱應該足以描述性，以很好地瞭解變數的用途。

Class Data Members

每個 class data member（也稱為 instance variable 或 member variable）的目的必須明確。 如果有任何不變數（特殊值、member 之間的關係、lifetime）沒有明確由型別和名稱表示，則必須對其進行評論。 但是，如果型別和名稱足夠（ `int num_events_;` ），則無需註釋。

特別說明，需新增註釋來描述特殊值的存在和含義，如 `nullptr` 或 `-1` 。 例如：

```
private:
 // Used to bounds-check table accesses. -1 means
 // that we don't yet know how many entries the table has.
 int num_total_entries_;
```

Global Variables

所有全域性變數都應該有一個註釋，描述它們是什麼，它們用於什麼，以及（如果不清楚）為什麼它們需要全域性。 例如：

```
// The total number of test cases that we run through in this regression test.
const int kNumTestCases = 6;
```

69. Implementation Comments

在您的 implementation 中，您應該在程式的棘手、不明顯、有趣或重要部分註釋。

Explanatory Comments

棘手或複雜的 code blocks 應該有註釋 ，並且寫在前面。

Function Argument Comments

當函式引數的含義不明顯時，請考慮以下補救措施之一：

- 如果引數是一個字面常量，並且在多個函式呼叫中使用相同的常量，而預設它們是相同的，您應該使用命名常量來明確該約束，並保證它成立。
- 考慮更改函式定義，將 `bool` parameter 替換為 enums。 這將使參數 self-describing。
- 對於具有多個配置選項的函式，請考慮定義單個 class 或 struct 來儲存所有選項，並傳遞其中一個 instance。 這種方法有幾個優點。 選項在變數命名就可以反應它們的含義。 它還減少了函式 argument count，這使得函式呼叫更容易讀寫。 作為一項額外的好處，當您新增其他選項時，您不必更改程式碼。
- 用命名新變數替換大型或複雜的 nested expressions。

- 作為最後手段，在 call 的地方使用註解來說明含義。

考慮以下示例：

```
// What are these arguments?
const DecimalNumber product = CalculateProduct(values, 7, false, nullptr);


ProductOptions options;
options.set_precision_decimals(7);
options.set_use_cache(ProductOptions::kDontUseCache);
const DecimalNumber product =
    CalculateProduct(values, options, /*completion_callback=*/nullptr);
```

Don'ts

不要陳述顯而易見的邏輯。 特別是，不要從字面上描述程式的作用，除非該行為對理解 C++ 的讀者來說並不明顯。 相反，提供更高級別的註釋，描述為什麼程式會做它所做的事情，或者讓程式 self describing。

比較這個：

```
// Find the element in the vector.  <-- Bad: obvious!
if (std::find(v.begin(), v.end(), element) != v.end()) {
  Process(element);
}


// Process "element" unless it was already processed.
if (std::find(v.begin(), v.end(), element) != v.end()) {
  Process(element);
}
```

Self-describing 程式不需要註解。 下面示例中的註解是顯而易見的：

```
if (!IsAlreadyProcessed(element)) {
  Process(element);
}
```

70. Punctuation, Spelling, and Grammar

注意標點符號、拼寫和語法。

評論應該像敘事文字一樣可讀，並有適當的大寫和標點符號。 在許多情況下，完整的句子比句子片段更可讀。 較短的評論，例如一行程式末尾的評論，有時可能不那麼正式，但您應該與您的風格保持一致。

71. TODO Comments

對於臨時、短期解決方案或足夠但不完美的程式，請使用 `TODO` 註釋。

`TODO` 這個標籤應全部大寫，並用小括號包含自己的名字，如：

```
// TODO(John): Use a "\*" here for concatenation operator.
```

如果您的 `TODO` 是“在未來做某事”的形式，請確保您包含一個非常特定的日期（“在2005年11月之前修復”）或一個非常特定的事件（“當所有客戶端都可以處理 XML responses 時，請刪除此程式”）。