Google C++

..... 4
1. #define 4
2. 4
3. 5
4. -inl.h 5
5. Function Parameter Ordering 5
6. 6
..... 7
1. Namespaces 7
2. Nested Class 9
3. Nonmember Static Member Global Functions 9
4. Local Variables 10
5. Global Variables 10
..... 11
1. Constructor 11
2. Default Constructors 12
3. Explicit Constructors 12
4. Copy Constructors 13
5. Structs vs. Classes 14
6. Inheritance 14
7. Multiple Inheritance 15
8. Interface 15
9. Operator Overloading 16
10. Access Control 16
11. Declaration Order 17
12. Write Short Functions 17
Google 18
1. Smart Pointers 18
C++ 19
1. Reference Arguments 19
2. Function Overloading 19
3. Default Arguments 20
4. alloca Variable-Length Arrays and alloca() 20
5. Friends 20
6. Exceptions 20
7. Run-Time Type Information, RTTI 22
8. Casting 22

9. Streams 23
10. Preincrement and Predecrement 24
11. const Use of const 24
12. Integer Types 25
13. 64 64-bit Portability 26
14. Preprocessor Macros 27
15. 0 NULL 0 and NULL 27
16. sizeof sizeof 28
17. Boost Boost 28
.....29
1. General Naming Rules 29
2. File Names 30
3. Type Names 31
4. Variable Names 31
5. Constant Names 31
6. Function Names 32
7. Namespace Names 32
8. Enumerator Names 32
9. Macro Names 33
10. Exceptions to Naming Rules 33
.....34
1. Comment Style 34
2. File Comments 34
3. Class Comments 34
4. Function Comments 35
5. Variable Comments 36
6. Implementation Comments 37
7. Punctuation, Spelling and Grammar 38
8. TODO TODO Comments 38
.....39
1. Line Length 39
2. ASCII Non-ASCII Characters 40
3. Spaces vs. Tabs 40
4. Function Declarations and Definitions 40
5. Function Calls 42
6. Conditionals 43
7. Loops and Switch Statements 44
8. Pointers and Reference Expressions 45
9. Boolean Expressions 46
10. Return Values 46
11. Variable and Array Initialization 46
12. Preprocessor Directives 46
13. Class Format 47
14. Initializer Lists 48

15. Namespace Formatting 48
16. Horizontal Whitespace 49
17. Vertical Whitespace 50
.....52
1. Existing Non-conformant Code 52
2. Windows Windows Code 52
.....53

.cc C++ .h

main() .cc

1. #define

#define multiple inclusion

*<PROJECT>_<PATH>_<FILE>_H_*

foo

`foo/src/bar/baz.h`

```
#ifndef FOO_BAR_BAZ_H_
#define FOO_BAR_BAZ_H_
...
#endif // FOO_BAR_BAZ_H_
```

2.

forward declarations .h #include

dependency

.cc

3.

10 inline function

Definition

accessor mutator

instruction cache

10

switch

switch

4. -inl.h

-inl.h

.cc .h

.h

.h -inl.h

-inl.h

-inl.h

-inl.h #define

5. Function Parameter Ordering

C/C++

const references /

non-const pointers

/ /

6.

hidden dependencies

C C++ .h .h

UNIX .

.. google-awesome-project/src/base/logging.h

#include "base/logging.h"

dir/foo.cc dir2/foo2.h foo.cc

dir2/foo2.h

C

C++

.h .cc

dir/foo.cc dir2/foo2.h base/basictypes_unittest.cc

base/basictypes.h

google-awesome-project/src/foo/internal/fooserver.cc

```
#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"
```

1. Namespaces

.cc unnamed namespaces
:-(
using

name axis

Foo
project1::Foo project2::Foo

C++ One Definition Rule (ODR)

1) Unnamed Namespaces

.cc

```
namespace {                          // .cc

//
enum { UNUSED, EOF, ERROR };         //
bool AtEof() { return pos_ == EOF; }  //                    EOF

}  // namespace
```

//
namespace

.h

2) Named Namespaces

/

```
// .h
namespace mynamespace {
```

```
//
//
class MyClass {
public:
  ...
  void Foo();
};

}  // namespace mynamespace

// .cc
namespace mynamespace {

//
void MyClass::Foo() {
  ...
}

}  // namespace mynamespace
```

.cc

```
#include "a.h"

DEFINE_bool(someflag, false, "dummy flag");

class C;  //                    C
namespace a { class A; }  //          a        a::A

namespace b {

...code for b...              // b

}  // namespace b
```

std std

using

```
//      ——
using namespace foo;
```

.cc .h using

```
//         .cc
// .h
using ::foo::bar;
```

.cc .h

```
//          .cc
// .h
namespace fbz = ::foo::bar::baz;
```

2. Nested Class

member class

```
class Foo {

private:
  // Bar          Foo
  class Bar {
    ...
  };

};
```

enclosing class

.cc

Foo::Bar*

Foo

public

3. Nonmember Static Member Global Functions

.cc

static `static int Foo() {...}`

4. Local Variables

C++

+

```
int i;
i = f();        //    ——
nt j = g();     //    ——
```

gcc for (int i = 0; i < 10; ++i) i for

for i if while scope declaration

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

```
//
for (int i = 0; i < 1000000; ++i) {
  Foo f;  //                    1000000
  f.DoSomething(i);
}
```

```
Foo f;  //                    1
for (int i = 0; i < 1000000; ++i) {
  f.DoSomething(i);
}
```

5. Global Variables

class

bugs

class STL string, vector

class singleton pattern

C STL

const char kFrogSays[] = "ribbet";

.cc

class

---

1. .cc using

2. public

3.

4. class STL bugs

C++

1. Constructor

trivial

“ ”

Init() non-trivial

1)

2)

3)

4) main()
gflags

non-trivial Init()

2. Default Constructors

new[]

“ ”

internal state

3. Explicit Constructors

C++ explicit

conversion
Foo::Foo(string name) Foo
Foo::Foo(string name)
Foo

explicit

explicit

explicit Foo(string name);

explicit

4. Copy Constructors

DISALLOW_COPY_AND_ASSIGN

STL

C++ bugs

assignment operator public

private dummy private DISALLOW_COPY_AND_ASSIGN

```
//
//        private:
#define DISALLOW_COPY_AND_ASSIGN(TypeName) \
  TypeName(const TypeName&);               \
  void operator=(const TypeName&)

class Foo {
public:
  Foo(int f);
  ~Foo();

private:
  DISALLOW_COPY_AND_ASSIGN(Foo);
};
```

DISALLOW_COPY_AND_ASSIGN

operator= self-assignment

STL
STL std::tr1::shared_ptr

5. Structs vs. Classes

struct class

C++ struct class

struct passive objects

Initialize() Reset()
Validate()

class class

STL functors traits class struct

6. Inheritance

composition GoF Design Patterns

C++
implementation inheritance
interface inheritance

compile-time declaration
API API

physical layout

public

“ ” "is-a"
"has-a" Bar “ ”Foo Bar
Foo

virtual

protected

virtual virtual

7. Multiple Inheritance

multiple implementation inheritance

Interface

superclass

Interface

Windows

8. Interface

Interface

1) "=0"

2)

3) protected

4) Interface

1

Stroustrup The C++ Programming Language, 3rd edition 12.4

Interface

Java

Interface

Interface

Interface

9. Operator Overloading

+ /

int Equals()

Add()

1)

2) Equals() ==

3) bugs Foo + 4 &Foo + 4

4) &

operator=

Equals() CopyFrom()

“ ”C++

operator<<(ostream&, const T&)

STL key operator==

operator<

STL operator==

10. Access Control

foo_ foo()

set_foo()

11. Declaration Order

public: private:

public: protected: private:

1) typedefs enums

2)

3)

4)

5)

6)

DISALLOW_COPY_AND_ASSIGN private:

.cc

12. Write Short Functions

40

bugs

1.

2.

3. **explicit**

4. **private**

5. **struct**

6. **>** **>** **>** **virtual**

7.

8. **Interface**
protected

9.

10.

11. **public->protected->private**

12.

Google

Google C++ C++

1. Smart Pointers

scoped_ptr
STL std::tr1::shared_ptr
auto_ptr

“ ” scoped_ptr scoped_ptr
shared_ptr shared_ptr
reference-counting

field

Google **:D**

C++

1. Reference Arguments

const

C parameter int foo(int *pval) C++ int foo(int &val)

(*pval)++

NULL

const

```
void Foo(const string &in, string *out);
```

bind2nd mem_fun STL

AppendString()
AppendInt() Append()

3. Default Arguments

API

API

4. alloca Variable-Length Arrays and alloca()

alloca()

alloca()

alloca() C++ stack
“
”

allocator scoped_ptr/scoped_array

5. Friends

FooBuilder Foo FooBuilder
Foo

public

6. Exceptions

C++

1) “ ”

2) C++ Python Java C++

3) C++

4) factory function
Init() “ ”

5) testing framework

1) throw
if f() calls g() calls h()
h f g

2) control flow

3) RAII

4)

5)

:-(

Google C++

Google

Google Google

Windows :D

7. Run-Time Type Information, RTTI

RTTI

RTTI C++

RTTI

RTTI

Visitor

RTTI

RTTI RTTI-like workaround RTTI

:D

8. Casting

static_cast<>() C++ int y = (int)x int y = int(x);

C++ C

C (int)3.5

(int)"hello" C++

nasty

C++ C

1) static_cast C

2) const_cast const

3) reinterpret_cast

4) dynamic_cast

RTTI

9. Streams

printf() scanf()

gcc printf

pread() printf

%.*s %1s

printf

Only One Way

I/O I/O

printf + read/write

```
cout << this;  // Prints the address
cout << *this;  // Prints the contents
```

<<

printf

```
cerr << "Error connecting to '" << foo->bar()->hostname.first
     << ":" << foo->bar()->hostname.second << ": " << strerror(errno);

fprintf(stderr, "Error connecting to '%s:%u: %s",
        foo->bar()->hostname.first, foo->bar()->hostname.second,
```

strerror(errno));
“ ”

“ ”
printf + read/write

10. Preincrement and Predecrement

++i

++i i++ --i i--

++i i++
i i

C for
i ++

11. const Use of const

const

const const int
foo const class Foo
{ int Bar(char c) const; };

const const
const_cast

const
const

1) const

2) const const
const const const

3) const

const const int * const * const x;
x const int** x

mutable

const

int const *foo const int* foo
const int
" " const
const int

const

12. Integer Types

C++ int <stdint.h>
precise-width int16_t

C++ short 16 int 32 long 32
long long 64

C++

<stdint.h> int16_t uint32_t int64_t
short unsigned long long C int
size_t ptrdiff_t

int
int 32 32 64
int64_t uint64_t

int64_t

uint32_t bit pattern
assertion

C bugs

```
for (unsigned int i = foo.Length()-1; i >= 0; --i) ...
```

gcc bug bug
C type-promotion
scheme C

13. 64 64-bit Portability

64 32
structure alignment

1) printf() 32 64 C99
MSVC 7.1
inttypes.h

```
// printf macros for size_t, in the style of inttypes.h
#ifdef _LP64
#define __PRIS_PREFIX "z"
#else
#define __PRIS_PREFIX
#endif

// Use these macros after a % in a printf format string
// to get correct 32/64 bit behavior, like this:
// size_t size = records.size();
// printf("%"PRIuS"\n", size);

#define PRIdS __PRIS_PREFIX "d"
#define PRIxS __PRIS_PREFIX "x"
#define PRIuS __PRIS_PREFIX "u"
#define PRIXS __PRIS_PREFIX "X"
#define PRIoS __PRIS_PREFIX "o"
```

void *	%lx	%p		
int64_t	%qd, %lld	%"PRId64"		
uint64_t	%qu, %llu, %llx	%"PRIu64", %"PRIx64"		
size_t	%u	%"PRIuS", %"PRIxS"	C99	%zu
ptrdiff_t	%d	%"PRIdS"	C99	%zd

PRI*
PRI* %

printf("x = %30"PRIuS"\n", x) 32 Linux printf("x = %30" "u" "\n", x) printf("x = %30u\n", x)

2) sizeof(void *) != sizeof(int) intptr_t

3) 64
int64_t/uint64_t / 8 32 64

gcc __attribute__((packed)) MSVC #pragma pack() __declspec(align())

4) 64 LL ULL

```
int64_t my_value = 0x123456789LL;
uint64_t my_mask = 3ULL << 48;
```

5) 32 64

14. Preprocessor Macros

C++ C performance-critical code const “ ” …… #define

stringifying # concatenation ##

1) .h

2) #define #undef

3) #undef

4) C++ unbalanced C++ constructs

15. 0 NULL 0 and NULL

0 0.0 NULL '\0'

0 0.0

0 NULL Bjarne Stroustrup 0
NULL C++ gcc 4.1.0
NULL sizeof(NULL) sizeof(0)

'\0'

16. sizeof sizeof

sizeof(*varname*) sizeof(*type*)

sizeof(*varname*) sizeof(*type*)

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

17. Boost Boost

Boost

Boost peer-reviewed
C++

Boost C++
type traits binders TR1

Boost metaprogramming
“ ” "functional"

Boost

1) Compressed Pair boost/compressed_pair.hpp

2) Pointer Container boost/ptr_container ptr_array.hpp
serialization

Boost

C++

1. scoped_ptr

2. const

3.

4.

5.

6.

7.

8. RTTI

9. C++ dynamic_cast

10. printf + read/write it is a problem

11. / /

12. const const

13.

14. 32 64

15.

16. 0 0.0 NULL '\0'

17. sizeof(varname) sizeof(type)

18. Boost

1. General Naming Rules

" "

```
int num_errors;                  // Good.
int num_completed_connections;   // Good.
```

```
int n;                          // Bad - meaningless.
int nerr;                       // Bad - ambiguous abbreviation.
int n_comp_conns;               // Bad - ambiguous abbreviation.
                      FileOpener  num_errors

                    OpenFile()  set_num_errors()




// Good
// These show proper names with no abbreviations.
int num_dns_connections;  // Most people know what "DNS" stands for.
int price_count_reader;   // OK, price count. Makes sense.

// Bad!
// Abbreviations can be confusing or ambiguous outside a small group.
int wgc_connections;  // Only your group knows what this stands for.
int pc_reader;        // Lots of things can be abbreviated "pc".


int error_count;  // Good.
int error_cnt;    // Bad.
```

2. File Names

_ -

my_useful_class.cc
my-useful-class.cc
myusefulclass.cc

C++ .cc .h

/usr/include UNIX Linux

db.h

http_server_logs.h logs.h

foo_bar.h foo_bar.cc FooBar

.h .h

-inl.h

```
url_table.h      // The class declaration.
```

```
url_table.cc     // The class definition.
url_table-inl.h  // Inline functions that include lots of code.
```

[redacted]-inl.h

3. Type Names

MyExcitingClass MyExcitingEnum

—— typedef ——

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

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

// enums
enum UrlTableErrors { ...
```

4. Variable Names

my_exciting_local_variable my_exciting_member_variable_

```
string table_name;  // OK - uses underscore.
string tablename;   // OK - all lowercase.
string tableName;   // Bad - mixed case.
```

```
struct UrlTableProperties {
  string name;
  int num_entries;
}
```

[redacted] **vs.**

g_

5. Constant Names

k kDaysInAWeek

k

```
const int kDaysInAWeek = 7;
```

6. Function Names

regular functions

accessors and mutators MyExcitingFunction() MyExcitingMethod() my_exciting_member_variable() set_my_exciting_member_variable()

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

num_entries_

```
class MyClass {
 public:
  ...
  int num_entries() const { return num_entries_; }
  void set_num_entries(int num_entries) { num_entries_ = num_entries; }

 private:
  int num_entries_;
};
```

7. Namespace Names

google_awesome_project

8. Enumerator Names

MY_EXCITING_ENUM_VALUE

UrlTableErrors

```
enum UrlTableErrors {
  OK = 0,
  ERROR_OUT_OF_MEMORY,
```

```
    ERROR_MALFORMED_INPUT,
  };
```

9. Macro Names

MY_MACRO_THAT_SCARES_SMALL_CHILDREN

```
  #define ROUND(x) ...
  #define PI_ROUNDED 3.0
  MY_EXCITING_ENUM_VALUE
```

10. Exceptions to Naming Rules

C/C++

bigopen()

open()

uint

typedef

bigpos

struct class pos

sparse_hash_map

STL STL

LONGLONG_MAX

INT_MAX

---

1. ChangeLocalValue ChgLocVal

ModifyPlayerName MdfPlyNm

2. +

3. +

g_

4.

5.

self-documenting

1. Comment Style

// /* */

// /* */ //

2. File Comments

1) copyright statement Copyright 2008 Google Inc.

2) license boilerplate Apache 2.0 BSD LGPL GPL

3) author line

.h .cc .cc .h .cc .h

.h .cc

3. Class Comments

```
// Iterates over the contents of a GargantuanTable.  Sample usage:
//    GargantuanTable_Iterator* iter = table->NewIterator();
//    for (iter->Seek("foo"); !iter->done(); iter->Next()) {
//      process(iter->key(), iter->value());
//    }
```

```
//    delete iter;
class GargantuanTable_Iterator {
  ...
};
```

“ ”

synchronization assumptions

4. Function Comments

"Opens the file"

"Open the file"

1) inputs outputs

2)

3)

4) NULL

5) performance implications

6) re-entrant synchronization assumptions

```
// Returns an iterator for this table.  It is the client's
// responsibility to delete the iterator when it is done with it,
// and it must not use the iterator once the GargantuanTable object
// on which the iterator was created has been deleted.
//
// The iterator is initially positioned at the beginning of the table.
//
// This method is equivalent to:
//    Iterator* iter = table->NewIterator();
//    iter->Seek("");
//    return iter;
```

```
// If you are going to immediately seek to another place in the
// returned iterator, it will be faster to use NewIterator()
// and avoid the extra seek.
Iterator* GetIterator() const;
```

"returns false otherwise"

```
// Returns true if the table cannot hold any more entries.
bool IsTableFull();
```

/ / "destroys this object"

.h

5. Variable Comments

NULL -1

sentinel values

```
private:
 // Keeps track of the total number of entries in the table.
 // Used to ensure we do not go over the limit. -1 means
 // that we don't yet know how many entries the table has.
 int num_total_entries_;
```

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

6. Implementation Comments

```
// Divide result by two, taking into account that x
// contains the carry from the add.
for (int i = 0; i < result->size(); i++) {
  x = (x << 8) + (*result)[i];
  (*result)[i] = x >> 1;
  x &= 1;
}
```

```
// If we have enough memory, mmap the data portion too.
mmap_budget = max<int64>(0, mmap_budget - index_->length());
if (mmap_budget >= data_size_ && !MmapData(mmap_chunk_bytes, mlock))
  return;  // Error already logged.
```

```
...
DoSomething();                  // Comment here so the comments line up.
DoSomethingElseThatIsLonger();  // Comment here so there are two spaces between
                                // the code and the comment.
...
```

NULL true/false 1 2 3……

```
bool success = CalculateSomething(interesting_value,
                                  10,
                                  false,
                                  NULL);  // What are these arguments??
```

```
bool success = CalculateSomething(interesting_value,
                                  10,     // Default base value.
```

```
                                false,  // Not the first time we're calling
this.
                                NULL);  // No callback.
```

```
const int kDefaultBaseValue = 10;
const bool kFirstTimeCalling = false;
Callback *null_callback = NULL;
bool success = CalculateSomething(interesting_value,
                                  kDefaultBaseValue,
                                  kFirstTimeCalling,
                                  null_callback);
```

C++ :D

```
// Now go through the b array and make sure that if i occurs,
// the next element is i+1.
...        // Geez.  What a useless comment.
```

7. Punctuation, Spelling and Grammar

.

semicolon comma

8. TODO TODO Comments

TODO

TODO parentheses
colon TODO

```
// TODO(kl@gmail.com): Use a "*" here for concatenation operator.
// TODO(Zeke) change this to use relations.
```

" " "Fix by November 2005" "Remove this code when all clients can handle XML responses."

---

1. C++ coders C coders

2.

3.

4. Chinese coders it is a problem

5.
UNIX/LINUX tab space
space

6. TODO

1. Line Length

80

80

80 60

80

1) 80 URL 80

2) 80

3)

2. ASCII Non-ASCII Characters

ASCII UTF-8

ASCII

ASCII

ASCII UTF-8

—— "\xEF\xBB\xBF" Unicode zero-width no-break space UTF-8

3. Spaces vs. Tabs

2

tabs tab

4. Function Declarations and Definitions

```
ReturnType ClassName::FunctionName(Type par_name1, Type par_name2) {
  DoSomething();
  ...
}
```

```
ReturnType ClassName::ReallyLongFunctionName(Type par_name1,
                                             Type par_name2,
                                             Type par_name3) {
  DoSomething();
  ...
}
```

```
ReturnType LongClassName::ReallyReallyReallyLongFunctionName(
    Type par_name1,  // 4 space indent
    Type par_name2,
    Type par_name3) {
  DoSomething();  // 2 space indent
  ...
}
```

1)

2) open parenthesis

3)

4)

5) open curly brace

6) close curly brace

7) close parenthesis

8)

9)

10) 2

11) 4

const const

```
// Everything in this function signature fits on a single line
ReturnType FunctionName(Type par) const {
  ...
}

// This function signature requires multiple lines, but
// the const keyword is on the line with the last parameter.
ReturnType ReallyLongFunctionName(Type par1,
                                  Type par2) const {
  ...
}
```

```
// Always have named parameters in interfaces.
class Shape {
 public:
  virtual void Rotate(double radians) = 0;
}

// Always have named parameters in the declaration.
class Circle : public Shape {
 public:
  virtual void Rotate(double radians);
}
```

```
// Comment out unused named parameters in definitions.
void Circle::Rotate(double /*radians*/) {}
// Bad - if someone wants to implement later, it's not clear what the
// variable means.
void Circle::Rotate(double) {}
```

UNIX/Linux **.cc**

Windows

5. Function Calls

```
bool retval = DoSomething(argument1, argument2, argument3);
```

```
bool retval = DoSomething(averyveryveryverylongargument1,
                          argument2, argument3);
```

```
bool retval = DoSomething(argument1,
                          argument2,
                          argument3,
                          argument4);
```

```
if (...) {
  ...
  ...
  if (...) {
    DoSomethingThatRequiresALongFunctionName(
        very_long_argument1,  // 4 space indent
        argument2,
        argument3,
        argument4);
  }
```

6. Conditionals

else

```
if (condition) {  // no spaces inside parentheses
  ...  // 2 space indent.
} else {  // The else goes on the same line as the closing brace.
  ...
}
```

```
if ( condition ) {  // spaces inside parentheses - rare
  ...  // 2 space indent.
} else {  // The else goes on the same line as the closing brace.
  ...
}
```

if

```
if(condition)     // Bad - space missing after IF.
if (condition){   // Bad - space missing before {.
if(condition){    // Doubly bad.
if (condition) {  // Good - proper space after IF and before {.
```

else

```
if (x == kFoo) return new Foo();
if (x == kBar) return new Bar();
```

else

```
// Not allowed - IF statement on one line when there is an ELSE clause
if (x) DoThis();
else DoThat();
```

if

```
if (condition)
```

```
    DoSomething();  // 2 space indent.

  if (condition) {
    DoSomething();  // 2 space indent.
  }



  // Not allowed - curly on IF but not ELSE
  if (condition) {
    foo;
  } else
    bar;

  // Not allowed - curly on ELSE but not IF
  if (condition)
    foo;
  else {
    bar;
  }

  // Curly braces around both IF and ELSE required because
  // one of the clauses used braces.
  if (condition) {
    foo;
  } else {
    bar;
  }
```

7. Loops and Switch Statements

switch {} continue

switch case

case default case

default assert

```
  switch (var) {
    case 0: {  // 2 space indent
      ...      // 4 space indent
      break;
    }
    case 1: {
      ...
```

```
    break;
  }
  default: {
    assert(false);
  }
}
```

{} continue

```
while (condition) {
  // Repeat test until it returns false.
}
for (int i = 0; i < kSomeNumber; ++i) {}  // Good - empty body.
while (condition) continue;  // Good - continue indicates no logic.
while (condition);  // Bad - looks like part of do/while loop.
```

8. Pointers and Reference Expressions

. -> / * &

```
x = *p;
p = &x;
x = r.y;
x = r->y;
```

1)

2) * &

```
// These are fine, space preceding.
char *c;
const string &str;

// These are fine, space following.
char* c;    // but remember to do "char* c, *d, *e, ...;"!
const string& str;
char * c;  // Bad - spaces on both sides of *
const string & str;  // Bad - spaces on both sides of &
```

9. Boolean Expressions

80

&&

```
if (this_one_thing > this_other_thing &&
    a_third_thing == a_fourth_thing &&
    yet_another & last_one) {
  ...
}
```

&&

10. Return Values

return

```
return x;  // not return(x);
```

11. Variable and Array Initialization

= ()

```
int x = 3;
int x(3);
string name("Some Name");
string name = "Some Name";
```

12. Preprocessor Directives

```
// Good - directives at beginning of line
  if (lopsided_score) {
#if DISASTER_PENDING      // Correct -- Starts at beginning of line
    DropEverything();
#endif
```

```
      BackToNormal();
    }
  // Bad - indented directives
    if (lopsided_score) {
      #if DISASTER_PENDING  // Wrong!  The "#if" should be at beginning of
  line
      DropEverything();
      #endif                // Wrong!  Do not indent "#endif"
      BackToNormal();
    }
```

13. Class Format

public: protected: private: 1

private

:-)

```
class MyClass : public OtherClass {
 public:      // Note the 1 space indent!
  MyClass();  // Regular 2 space indent.
  explicit MyClass(int var);
  ~MyClass() {}

  void SomeFunction();
  void SomeFunctionThatDoesNothing() {
  }

  void set_some_var(int var) { some_var_ = var; }
  int some_var() const { return some_var_; }

 private:
  bool SomeInternalFunction();

  int some_var_;
  int some_other_var_;
  DISALLOW_COPY_AND_ASSIGN(MyClass);
};
```

1) 80

2) public: protected: private: 1 MSVC tab

3) public

4)

5) public protected private

6)

14. Initializer Lists

```
// When it all fits on one line:
MyClass::MyClass(int var) : some_var_(var), some_other_var_(var + 1) {


// When it requires multiple lines, indent 4 spaces, putting the colon on
// the first initializer line:
MyClass::MyClass(int var)
    : some_var_(var),             // 4 space indent
      some_other_var_(var + 1) {  // lined up
  ...
  DoSomething();
  ...
}
```

15. Namespace Formatting

```
namespace {

void foo() {  // Correct.  No extra indentation within namespace.
  ...
}

}  // namespace


namespace {
```

```
  // Wrong.  Indented when it should not be.
  void foo() {
    ...
  }

}  // namespace
```

16. Horizontal Whitespace

```
void f(bool b) {  // Open braces should always have a space before them.
  ...
int i = 0;  // Semicolons usually have no space before them.
int x[] = { 0 };  // Spaces inside braces for array initialization are
int x[] = {0};    // optional.  If you use them, put them on both sides!
// Spaces around the colon in inheritance and initializer lists.
class Foo : public Bar {
 public:
  // For inline function implementations, put spaces between the braces
  // and the implementation itself.
  Foo(int b) : Bar(), baz_(b) {}  // No spaces inside empty braces.
  void Reset() { baz_ = 0; }  // Spaces separating braces from
implementation.
  ...
```

```
if (b) {          // Space after the keyword in conditions and loops.
} else {          // Spaces around else.
}
while (test) {}   // There is usually no space inside parentheses.
switch (i) {
for (int i = 0; i < 5; ++i) {
switch ( i ) {    // Loops and conditions may have spaces inside
if ( test ) {     // parentheses, but this is rare.  Be consistent.
for ( int i = 0; i < 5; ++i ) {
for ( ; i < 5 ; ++i) {  // For loops always have a space after the
  ...                   // semicolon, and may have a space before the
                        // semicolon.
```

```
switch (i) {
  case 1:         // No space before colon in a switch case.
    ...
  case 2: break;  // Use a space after a colon if there's code after it.
```

```
x = 0;              // Assignment operators always have spaces around
                    // them.
x = -5;             // No spaces separating unary operators and their
++x;                // arguments.
if (x && !y)
  ...
v = w * x + y / z;  // Binary operators usually have spaces around them,
v = w*x + y/z;      // but it's okay to remove spaces around factors.
v = w * (x + z);    // Parentheses should have no spaces inside them.
```

```
vector<string> x;           // No spaces inside the angle
y = static_cast<char*>(x);  // brackets (< and >), before
                            // <, or between >( in a cast.
vector<char *> x;           // Spaces between type and pointer are
                            // okay, but be consistent.
set<list<string> > x;       // C++ requires a space in > >.
set< list<string> > x;      // You may optionally make use
                            // symmetric spacing in < <.
```

17. Vertical Whitespace

2

```
void Function() {

  // Unnecessary blank lines before and after

}
```

```
   while (condition) {
     // Unnecessary blank line after

   }
   if (condition) {

     // Unnecessary blank line before
   }
```

if-else

```
   if (condition) {
     // Some lines of code too small to move to another function,
     // followed by a blank line.

   } else {
     // Another block of code
   }
```

---

1. 80 22

2. ASCII UTF-8 UNIX/Linux Windows

3. UNIX/Linux MSVC Tab

4.

5. / / /

6. ./-> */&

7. / / / / /

8. = ()

9. return ()

10. /

1. Existing Non-conformant Code

2. Windows Windows Code

Windows Windows Microsoft C++

Windows

:D

1) Hungarian notation iNum Google .cc

2) Windows DWORD HANDLE Windows API C++ const TCHAR * LPCTSTR

3) Microsoft Visual C++ **3 warnings errors**

4) #pragma once; C++ #include<prj_name/public/tools.h>

5) #pragma __declspec __declspec(dllimport) __declspec(dllexport) DLLIMPORT DLLEXPORT

Windows

1) COM ATL/WTL COM ATL/WTL

2) ATL STL Visual C++ STL ATL _ATL_NO_EXCEPTIONS STL STL

3) StdAfx.h precompile.h
precompile.cc
/FI

4) resource.h

if
*

```
/*********************************
* Some comments are here.
* There may be many lines.
*********************************/
```

Benjy Weinberger
Craig Silverstein
Gregory Eitzmann
Mark Mentovai
Tashana Landray