Guia de Estilo de Código C++ (Google)

Resumo de alguns casos relevantes do guia de estilo oficial

Neste documento tentam-se identificar algumas das partes mais relevantes do guia de estilo de código em C++ desenhado pela Google[[1]](#footnote-0). Este documento, sendo um resumo, **não descarta a leitura do documento oficial** onde a informação se encontra descrita de uma forma mais completa..

[**Ficheiros de Cabeçalho**](#_2tlofzs59uyj) **2**

[Declarações em avanço](#_gvmrqzsj5hz7) 2

[Funções inline](#_m2f26mn8v3ny) 2

[Nomes e ordem de inclusões](#_otl1x7kcdjzc) 2

[**Contextos de Existência (Scoping)**](#_d98c6n2t424b) **3**

[Variáveis locais](#_26g3xo7rtf8m) 3

[Variáveis globais e estáticas](#_z3kkdxaoaa2z) 3

[**Classes**](#_3v15ekpd6eiq) **4**

[Construtores](#_rqwrv2ioneqv) 4

[Estruturas de Dados vs. Classes](#_2vypq4nuxg8n) 4

[Herança](#_vx9gn0uogfxf) 4

[Overloading de operadores](#_2fuk29wniew4) 4

[Controlo de acesso](#_15wqly3yhuyv) 4

[Ordem de declaração](#_qpnz3v1s6v4u) 4

[**Funções**](#_8844yc631asi) **4**

[Entradas e saídas](#_x7wsqvndrmob) 4

[Overloading de funções](#_i5vnkzpa5kia) 5

[Argumentos com valor por defeito](#_oxanpb2ogwvk) 5

[**Outras funcionalidades do C++**](#_pz2gj83t3i7w) **5**

[Casting](#_xrlx1b8a7gji) 5

[Streams](#_82rprhp7ym0p) 5

[Pré-incremento e pré-decremento](#_ukgi3sualt46) 5

[Uso de qualificação const](#_3t9c8n3qg478) 5

[Uso de 0 e nullptr/NULL](#_4jjheknjuo06) 6

[Uso de sizeof](#_bi411p79xnsc) 6

[Dedução de tipos de dados (uso de auto)](#_y7gs2x7n12qc) 6

[**Nomenclatura**](#_s9cq1z2rysa) **6**

[Regras de nomeação](#_o8lq5th7h5a) 6

[Nomes de ficheiros](#_hra1cgsdn8mn) 7

[Nomes de tipos de dados](#_mhle9x9k5ndk) 7

[Nomes de variáveis](#_lcba5rxees8f) 7

[Nomes de constantes](#_hndc9g96j8pg) 8

[Nomes de funções](#_g43owmwe0gzh) 8

[Nomes de enumerados](#_wv3z9zggle2l) 8

[**Comentários**](#_h9hy1ms7hugm) **8**

[**Formatação**](#_5b6mdk1k4zdf) **8**

[Tamanho das linhas](#_dimxtyha1ayn) 8

[Carateres não-ASCII](#_wds8kks2bx2z) 9

[Espaços vs. Tabs](#_1d73bdjv7r0f) 9

[Declaração e definição de funções](#_t0mzr4k4ig04) 9

[Literais em vírgula flutuante](#_wuxpxuvn0nz3) 9

[Invocações de funções](#_eae3oexilylf) 10

[Condições](#_vumd705fbvja) 10

[Instruções de ciclo e switch](#_7losambq03gx) 11

[Expressões de ponteiro e referência](#_nc82tr1lnf49) 11

[Expressões booleanas](#_kc8jpusn2wp5) 12

[Valores de retorno](#_4a6hntu4lbn) 12

[Inicialização de variáveis e arrays](#_2rd0bc2gk4l1) 12

[Formato de classes](#_iq3y5bn78r2b) 12

[Espaços em branco na horizontal](#_gy17574bsdta) 13

[Espaços em branco na vertical](#_hk25be8a30y0) 14

## Ficheiros de Cabeçalho

### Declarações em avanço

Evitem-se fazer declarações sem implementação. Pelo contrário, incluam-se sempre os cabeçalhos que as contenham. Mais informação em [*link*](https://google.github.io/styleguide/cppguide.html#Forward_Declarations).

| Não fazer | Fazer |
| --- | --- |
| // In a C++ source file:  class B;  void FuncInB();  extern int variable\_in\_b;  ABSL\_DECLARE\_FLAG(flag\_in\_b); | // In a C++ source file:  #include "b.h" |

### Funções *inline*

Defina-se este tipo de funções somente se esta é pequena, ou seja, que ocupe, no máximo, 10 linhas. Mais informação em [*link*](https://google.github.io/styleguide/cppguide.html#Inline_Functions).

### Nomes e ordem de inclusões

Incluam-se os cabeçalhos na seguinte ordem: [cabeçalho] diretamente relacionado; do sistema C; das bibliotecas da norma do C++; outras bibliotecas; outras inclusões do projeto. Mais informação em [*link*](https://google.github.io/styleguide/cppguide.html#Names_and_Order_of_Includes).

**Exemplo:**

Em dir/foo.cpp ou em dir/foo\_test.cpp, em que se implementa ou testam tarefas em dir2/foo2.h, ordenem-se as nossas inclusões da seguinte forma:

1. dir2/foo.h
2. Linha vazia
3. Cabeçalhos do sistema C (cabeçalhos entre <> com a extensão .h, como <unistd.h> ou <stdlib.h>)
4. Linha vazia
5. Cabeçalhos das bibliotecas da norma do C++ (sem extensão de ficheiro), como <algorithm> ou <iostream>
6. Linha vazia
7. Outros cabeçalhos de ficheiros de bibliotecas com extensão .h, entre aspas
8. Linha vazia
9. Cabeçalhos de ficheiros .h do nosso projeto.

## Contextos de Existência (*Scoping*)

### Variáveis locais

Coloquem-se as declarações das variáveis no contexto de existência mais curto possível, inicializando-as sempre que possível a seguir à sua declaração. Mais informação em [*link*](https://google.github.io/styleguide/cppguide.html#Local_Variables).

|  | Não fazer | Fazer |
| --- | --- | --- |
| 1 | int i;  i = foo(); | int i = foo(); |
| 2 | std::vector<int> v;  v.push\_back(1);  v.push\_back(2); | std::vector<int> v = {1, 2}; |
| 3 | int i;  for (i = 0; i < 10; ++i) { ... } | for (int i = 0; i < 10; ++i) { ... } |

As variáveis que são necessárias para os contextos de if, while e for devem ser declaradas nos contextos referidos, por norma, confinadas a esses âmbitos, como se pode ver no caso 3 acima.

### Variáveis globais e estáticas

Objetos armazenados com duração estática são proibidos a não ser que sejam passíveis de serem trivialmente destruídos. Mais informação em [*link*](https://google.github.io/styleguide/cppguide.html#Static_and_Global_Variables).

## Classes

### Construtores

Evite-se invocar métodos virtuais em construtores e inicializações que podem falhar, no caso de não termos implementadas diretivas para tratamento de erros. Mais informação em [*link*](https://google.github.io/styleguide/cppguide.html#Doing_Work_in_Constructors).

### Estruturas de Dados *vs.* Classes

Use-se estruturas struct somente para objetos passivos que transportam dados; tudo o resto deverá ser caraterizado como class. Mais informação em [*link*](https://google.github.io/styleguide/cppguide.html#Structs_vs._Classes).

### Herança

Composição é, geralmente, mais apropriada que herança. Quando se usar herança, recomenda-se torná-la pública (public). Mais informação em [*link*](https://google.github.io/styleguide/cppguide.html#Inheritance).

### Overloading de operadores

Use-se o *overloading* de operadores com muito cuidado e atenção. Não se use literais definidos pelo utilizador. É fácil gerar confusão e *bugs* com esta tarefa implementada incorretamente. Mais informação em [*link*](https://google.github.io/styleguide/cppguide.html#Operator_Overloading).

### Controlo de acesso

Tornem-se todos os membros de uma classe como privados (private), a não ser que estes sejam constantes. Mais informação em [*link*](https://google.github.io/styleguide/cppguide.html#Access_Control).

### Ordem de declaração

Agrupem-se declarações consideradas semelhantes, juntas, deixando partes públicas no início destas. A definição de uma classe deverá portanto iniciar com uma secção pública (public:), seguida por uma protected: e uma private:. Caso haja uma ou mais destas secções que estejam vazias, omitam-se. Mais informação em [*link*](https://google.github.io/styleguide/cppguide.html#Declaration_Order).

## Funções

### Entradas e saídas

A saída de uma função em C++ é naturalmente fornecida através de um valor de retorno ou, por vezes, através de parâmetros (argumentos). Recomenda-se o uso de valores de retorno pela palavra return, aos argumentos, aumentando a legibilidade e, geralmente, dando o mesmo ou melhor desempenho.

Prefira-se, portanto, retornar por valor ou, na impossibilidade, retornar por referência. Evite-se passar por ponteiro a não ser que se este possa ter valor NULL. Mais informação em [*link*](https://google.github.io/styleguide/cppguide.html#Inputs_and_Outputs).

### Overloading de funções

Usem-se funções *overloaded* (incluindo construtores) somente se o leitor com uma rápida leitura perceber o intuito desta, sem ter de ir consultar ao certo que especificação de função é que está, de facto, a ser usada. Mais informação em [*link*](https://google.github.io/styleguide/cppguide.html#Function_Overloading).

class MyClass {

public:

void Analyze(const std::string &text);

void Analyze(const char \*text, size\_t textlen);

};

### Argumentos com valor por defeito

Argumentos com valor por defeito são permitidos em funções não-virtuais quando o valor por defeito é garantidamente sempre igual. Recomenda-se, por tanto, que se sigam as mesmas regras que em §§Overloading de funções. Mais informação em [*link*](https://google.github.io/styleguide/cppguide.html#Default_Arguments).

## Outras funcionalidades do C++

### Casting

Usem-se *casts* do C++ como os static\_cast<float>(double\_value), ou inicialização com chavetas para conversões de tipos aritméticos como em int64\_t y = int64\_t{1} << 42. Não usar *casts* com formatos como (int)x a não ser que seja para void.

O problema com *casts* em C é a ambiguidade que esta operação traz; por vezes podemos estar a fazer uma conversão (por exemplo, como em (int)3.5), e por vezes podemos estar a fazer um *cast*, ou seja, algo como (int)”hello”. As operações de *cast* em C++ e as inicializações com chavetas ajudam a evitar esta ambiguidade. Mais informação em [*link*](https://google.github.io/styleguide/cppguide.html#Casting).

### Streams

Usem-se *streams* quando for apropriado, manipuladas sempre de forma “simples”. Faça-se *overload* do operador << para cumprir funções de *streaming* só para tipos que representem valores, e escreva-se só o valor visível para o utilizador, sem detalhes de implementação. Mais informação em [*link*](https://google.github.io/styleguide/cppguide.html#Streams).

### Pré-incremento e pré-decremento

Use-se a forma de prefixo de incremento (++i) e decremento (--i), a não ser que seja importante a necessidade de diferir a semântica. Mais informação em [*link*](https://google.github.io/styleguide/cppguide.html#Preincrement_and_Predecrement).

### Uso de qualificação *const*

Em APIs, use-se a qualificação de const sempre que fizer sentido. Mais informação em [*link*](https://google.github.io/styleguide/cppguide.html#Use_of_const).

### Uso de 0 e nullptr/NULL

Use-se nullptr para ponteiros, e ‘\0’ para carateres (e não o literal 0). Mais informação em [*link*](https://google.github.io/styleguide/cppguide.html#0_and_nullptr/NULL).

### Uso de sizeof

Prefira-se o uso de sizeof para nomes de variáveis e não para tipos de dados. Mais informação em [*link*](https://google.github.io/styleguide/cppguide.html#sizeof).

| Não fazer | Fazer |
| --- | --- |
| MyStruct data;  memset(&data, 0, sizeof(MyStruct)); | MyStruct data;  memset(&data, 0, sizeof(data)); |

### Dedução de tipos de dados (uso de auto)

Use-se a dedução de tipos de dados só se fizer sentido para que o código se torne legível para leitores que possam não estar familiarizados com o projeto, ou se esta ação tornar o código mais seguro. Não usar meramente para evitar a inconveniência de se explicitar um tipo de dados. Mais informação em [*link*](https://google.github.io/styleguide/cppguide.html#Type_deduction).

## Nomenclatura

### Regras de nomeação

Otimize-se a escrita do código sempre tendo em conta a legibilidade do código, escolhendo sempre nomes que sejam claros para um leitor, mesmo que seja de uma equipa de desenvolvimento externa ao projeto. Mais informações em [*link*](https://google.github.io/styleguide/cppguide.html#General_Naming_Rules).

Usem-se nomes que descrevam o propósito ou intenção de um objeto. Sem que haja preocupações relativamente à poupança de espaço na horizontal, é mais importante escrever e organizar o código de forma a que este seja imediatamente compreendido por um novo leitor. Minimize-se o uso de abreviações que poderão ser desconhecidos para alguém externo ao projeto. Não se abrevie termos eclipsando letras dentro de uma palavra. Em termos globais, a descrição de um excerto de código deverá ser proporcional à visibilidade de um nome no seu contexto de existência. Por exemplo, n poderá ser um bom nome para uma função de 5 linhas, mas no contexto de uma classe provavelmente será muito vago.

| Não fazer | Fazer |
| --- | --- |
| class MyClass {  public:  int CountFooErrors(const  std::vector<Foo>& foos) {  int total\_number\_of\_foo\_errors = 0;  for (int foo\_index = 0; foo\_index < foos.size(); ++foo\_index) {  ...  ++total\_number\_of\_foo\_errors;  }  return total\_number\_of\_foo\_errors;  }  void DoSomethingImportant() {  int cstmr\_id = ...;  }  private:  const int kNum = ...;  }; | class MyClass {  public:  int CountFooErrors(const  std::vector<Foo>& foos) {  int n = 0;  for (const auto& foo : foos) {  ...  ++n;  }  return n;  }  void DoSomethingImportant() {  std::string fqdn = ...;  }  private:  const int kMaxAllowedConnections = ...; }; |

### Nomes de ficheiros

Os nomes de ficheiros devem ser todos em minúsculas e poderão incluir *underscores* (\_) ou traços (-). Siga-se a convenção que o projeto usa. Se não existir nenhum padrão local consistente, prefira-se o uso de *underscores*. Mais informação em [*link*](https://google.github.io/styleguide/cppguide.html#File_Names).

**Exemplos:**

* my\_useful\_class.cc
* my-useful-class.cc
* myusefulclass.cc
* myusefulclass\_test.cc

### Nomes de tipos de dados

Nomes de tipos de dados começam com letra maiúscula e têm letra maiúscula por cada nova palavra, sem *underscores*: MyExcitingClass, MyExcitingEnum.

Os nomes de todos os tipos — classes, estruturas, renomeações de tipos, enumerados, e parâmetros de tipo de *templates* — têm todos a regra de seguir a mesma convenção. Por exemplo:

// 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 { ...

### Nomes de variáveis

Os nomes das variáveis (incluindo parâmetros de funções) e membros de estruturas são sempre escritos em minúsculas, com *underscores* entre palavras. Membros de classes (mas não de estruturas) adicionalmente têm um *underscore* no fim do seu nome: a\_local\_variable, a\_struct\_data\_member, a\_class\_data\_member\_. Mais informação em [*link*](https://google.github.io/styleguide/cppguide.html#Variable_Names).

### Nomes de constantes

Variáveis declaradas como constantes e cujo valor é fixo durante a execução de um programa são nomeadas sempre com o k no início do nome, seguido de *mixed case*. Aqui os *underscores* podem ser usados como separadores nos casos raros em que as maiúsculas não sirvam o propósito. Mais informação em [*link*](https://google.github.io/styleguide/cppguide.html#Constant_Names).

const int kDaysInAWeek = 7;

const int kAndroid8\_0\_0 = 24; // Android 8.0.0

### Nomes de funções

Funções regulares são escritas com *mixed case*; funções de acesso e modificação podem ser nomeadas como variáveis. Assim sendo, as funções devem começar com letra maiúscula e ter uma maiúscula por cada nova palavra. Mais informação em [*link*](https://google.github.io/styleguide/cppguide.html#Function_Names).

AddTableEntry()

DeleteUrl()

OpenFileOrDie()

### Nomes de enumerados

Os enumerados devem ser nomeados como constantes. Mais informação em [*link*](https://google.github.io/styleguide/cppguide.html#Enumerator_Names).

| Não fazer | Fazer |
| --- | --- |
| enum class AlternateUrlTableError {  OK = 0,  OUT\_OF\_MEMORY = 1,  MALFORMED\_INPUT = 2,  }; | enum class UrlTableError {  kOk = 0,  kOutOfMemory,  kMalformedInput,  }; |

## Comentários

O estilo de comentário a seguir deverá ser com // ou com /\* \*/ conforme os intervenientes do projeto decidirem, desde que se crie uma convenção e que se mantenha a consistência. Mais informação em [*link*](https://google.github.io/styleguide/cppguide.html#Comments).

## Formatação

### Tamanho das linhas

Cada linha de texto no código deverá ter, no máximo, 80 carateres em tamanho. Embora controverso, já existe uma longa base de código na comunidade, e este critério ajuda a manter alguma consistência. Mais informações em [*link*](https://google.github.io/styleguide/cppguide.html#Line_Length).

### Carateres não-ASCII

O uso de carateres não-ASCII deve ser raro, e deverão seguir uma codificação UTF-8. Mais informação em [*link*](https://google.github.io/styleguide/cppguide.html#Non-ASCII_Characters).

### Espaços *vs.* Tabs

Use-se espaços, com indentações de 2 espaços de cada vez.

### Declaração e definição de funções

O tipo de retorno deverá constar na mesma linha que o nome da função e parâmetros, se possível.

ReturnType ClassName::FunctionName(Type par\_name1, Type par\_name2) {

DoSomething();

}

Se não for possível manter tudo numa linha só, listam-se os vários parâmetros de forma alinhada pelos tipos de dados.

ReturnType ClassName::ReallyLongFunctionName(Type par\_name1, Type par\_name2,

Type par\_name3) {

DoSomething();

}

Ou se nem o primeiro parâmetro couber no tamanho máximo de linha, listam-se os parâmetros na linha de baixo, alinhados pelo tipo de dados de cada parâmetro, com indentação de 4 espaços.

ReturnType LongClassName::ReallyReallyReallyLongFunctionName(

Type par\_name1, // 4 space indent

Type par\_name2,

Type par\_name3) {

DoSomething(); // 2 space indent

}

Mais informação em [*link*](https://google.github.io/styleguide/cppguide.html#Function_Declarations_and_Definitions).

### Literais em vírgula flutuante

Literais em vírgula flutuante deve ter sempre um separador decimal, com dígitos em ambos lados, mesmo que denotem notação exponencial. Mais informação em [*link*](https://google.github.io/styleguide/cppguide.html#Floating_Literals).

| Não fazer | Fazer |
| --- | --- |
| float f = 1.f;  long double ld = -.5L;  double d = 1248e6; | float f = 1.0f;  float f2 = 1; // Also OK  long double ld = -0.5L;  double d = 1248.0e6; |

## 

### Invocações de funções

Escrevam-se as funções numa só linha, ajustando a exposição dos vários argumentos dentro dos parêntesis conforme indicado em §§Declaração e definição de funções. Na ausência de outras considerações a tomar, use-se o número mínimo de linhas possível, incluindo colocar múltiplos argumentos em cada linha quando apropriado. Mais informações em [*link*](https://google.github.io/styleguide/cppguide.html#Function_Calls).

**Exemplo:**

bool result = DoSomething(argument1, argument2, argument3);

Ou caso não caibam todos os argumentos na mesma linha:

bool result = DoSomething(averyveryveryverylongargument1,

argument2, argument3);

Ou, em casos mais extremos, em linhas subsequentes com indentação a 4 espaços:

if (...) {

...

...

if (...) {

bool result = DoSomething(

argument1, argument2, // 4 space indent

argument3, argument4);

...

}

}

### Condições

Aquando da instrução de decisão if, incluindo as suas cláusulas opcionais de else if e else, coloque-se sempre um espaço entre o if e os parêntesis e as chavetas, mas sem espaçamentos entre os parêntesis e a condição ou inicializações. Se as inicializações opcionais estiverem presentes, deixe-se um espaço ou nova linha depois do ponto-e-vírgula, mas não antes. Mais informação em [*link*](https://google.github.io/styleguide/cppguide.html#Conditionals).

| Não fazer | Fazer |
| --- | --- |
| if(condition) {  if ( condition ) {  if (condition){  if(condition){  if (int a = f();a == 10) | if (condition) {  DoOneThing();  DoAnotherThing();  } else if (int a = f(); a != 3) {  DoAThirdThing(a);  } else {  DoNothing();  } |

## 

### Instruções de ciclo e *switch*

Instruções de decisão switch podem ou não usar chavetas para blocos. Anote-se casos com *fall-through* que possam não ser triviais. No que toca a ciclos com uma única linha no corpo de código é igualmente possível ocultar as chavetas. Corpos de ciclos vazios não devem incluir chavetas vazias ou, em alternativa, dever-se-á usar continue. Por exemplo, abaixo, podemos ver um caso em que a oclusão de corpo de ciclo faz a instrução parecer erradamente parte de um ciclo do…while.

| Não fazer | Fazer |
| --- | --- |
| while (condition); | while (condition) {  // Repeat test until it returns false.  }  for (int i = 0; i < kSomeNumber; ++i) {}  while (condition) continue; |

Blocos dentro de casos de uma instrução switch podem ter, ou não, chavetas, dependendo da nossa preferência. Mais informações em [*link*](https://google.github.io/styleguide/cppguide.html#Loops_and_Switch_Statements).

### Expressões de ponteiro e referência

Não fazer espaços entre pontos ou setas de acesso a membros de um objeto para onde uma referência ou ponteiro apontam. Operadores-ponto não têm espaços à direita. As seguintes expressões são exemplos de ponteiros e expressões de referência bem formatadas.

x = \*p;

p = &x;

x = r.y;

x = r->y;

Quando nos referimos a um ponteiro ou referência é possível colocar o espaço antes ou depois do asterisco/ampersand (\*/&, respetivamente). Mantenha-se sempre, contudo, consistência na escolha.

// These are fine, space preceding.

char \*c;

const std::string &str;

int \*GetPointer();

std::vector<char \*>

// These are fine, space following (or elided).

char\* c;

const std::string& str;

int\* GetPointer();

std::vector<char\*> // Note no space between '\*' and '>'

É permitido declarar múltiplas variáveis na mesma declaração, ainda que não se recomende caso algum destes ponteiros ou referências divirjam na sua representação. Mais informações em [*link*](https://google.github.io/styleguide/cppguide.html#Pointer_and_Reference_Expressions).

| Não fazer | Fazer |
| --- | --- |
| int x, \*y; // No & or \* in mult. declr.  int\* x, \*y; // Inconsistent spacing  char \* c; // Spaces on both sides of \*  const std::string & str; // Spaces on  both sides of & | int x, y; |

### Expressões booleanas

Quando temos uma expressão booleana que é longa (+80 carateres) há que ser consistente na forma como quebramos as linhas. No seguinte exemplo o operador lógico AND está sempre no fim das linhas. Mais informação em [*link*](https://google.github.io/styleguide/cppguide.html#Boolean_Expressions).

if (this\_one\_thing > this\_other\_thing &&

a\_third\_thing == a\_fourth\_thing &&

yet\_another && last\_one) {

...

}

### Valores de retorno

Não cobrir desnecessariamente os valores de retorno com parêntesis. Use-se parêntesis somente quando se usaria igualmente numa simples atribuição.

| Não fazer | Fazer |
| --- | --- |
| return (value);  return(result); | return result;  return (some\_long\_condition &&  another\_condition); |

### Inicialização de variáveis e arrays

Pode-se escolher entre =, () e {}; as seguintes linhas estão todas corretas. Mais informação em [*link*](https://google.github.io/styleguide/cppguide.html#Variable_and_Array_Initialization).

int x = 3;

int x(3);

int x{3};

std::string name = "Some Name";

std::string name("Some Name");

std::string name{"Some Name"};

### Formato de classes

As secções em public, protected e private deverão estar escritas nesta mesma ordem, cada uma indentada por um espaço.

O formato básico para uma definição de uma classe (faltando os comentários, cuja informação e detalhes deverão ser consultados em [*link*](https://google.github.io/styleguide/cppguide.html#Class_Comments)) é a seguinte:

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\_;

};

Questões a ter em conta:

* Qualquer nome de classe-base deverá estar na mesma linha que a subclasse, sujeito igualmente ao limite de 80 carateres;
* As *keywords* public:, protected:, e private: deverão ser indentadas por um espaço;
* À exceção da primeira instância, tais *keywords* deverão ser precedidas por uma linha em branco. Esta regra é opcional em classes pequenas;
* Não deixar uma linha em branco depois destas *keywords*;
* A secção public deverá constar primeiro, seguida da protected e depois da private.

### Espaços em branco na horizontal

O uso de espaço em branco na horizontal depende do local. Nunca colocar espaços à direita, ou seja, no fim das linhas. Mais informação em [*link*](https://google.github.io/styleguide/cppguide.html#Horizontal_Whitespace). No geral, siga-se o seguinte exemplo.

int i = 0; // Two spaces before end-of-line comments.

void f(bool b) { // Open braces should always have a space before them.

...

int i = 0; // Semicolons usually have no space before them.

// Spaces inside braces for braced-init-list are optional. If you use them,

// put them on both sides!

int x[] = { 0 };

int x[] = {0};

// 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.

...

### Espaços em branco na vertical

Minimize-se o uso de espaços em branco na vertical.

Isto é mais um princípio que uma regra: não se usem linhas em branco quando não precisarmos. Em particular, não se coloquem mais do que uma ou duas linhas em branco entre funções, resista-se à tentação de começar funções com uma linha em branco, assim como não se terminem funções com uma linha em branco. Uma linha em branco de código deve ter o mesmo propósito que um parágrafo em prosa: separa virtualmente dois pensamentos, duas lógicas.

O princípio mais básico é o seguinte: quanto mais código couber no ecrã, melhor será para o conseguir seguir e perceber o fluxo de controlo do programa. Mais informação em [*link*](https://google.github.io/styleguide/cppguide.html#Vertical_Whitespace).

1. https://google.github.io/styleguide/cppguide.html [↑](#footnote-ref-0)