

Guia de estilo Google C++

Tabela de Conteúdos

Versão C++	
Arquivos de cabeçalho	Cabeçalhos independentes #define Guard Incluso o que você usa Declarações de frente Inline Functions Names e Ordem de Incluir
Escopo	Namespaces Ação internal Nonmember , Membro Estático e Funções Globais Variáveis Estatáticas e Globais thread_local Variáveis
Classes	Fazendo trabalho em conversões de construtores ImplicitCopyable e Tipos Móveis conómicos vs. Classes Structs vs. Pares e Tuples Inheritance Operador Sobre carga SA ordem de decisão de declaração
Funções	Entradas e saídas SE Escreva funções curtas Função Sobrecarga Default Argumentos Trailing Return Type Syntax
Magia específica do Google	Propriedade e Ponteiros Inteligentes
Outros recursos C++	Referências Rvalue Friends Exceptions noexcept Run-Time Type Information (RTTI) Casting Streams Preincrement and Predecrement use const do constexpr Integer Types 64-bit Portability Macros do pré-processador e nullptr NULL Sizeof Type Dedução (incluindo auto) Dedução de argumento de modelo de classe Designated Initializers Lambda Expressions Template Metaprogramming Boost Other C++ Features Nostandard Extensions Aliases
Linguagem Inclusiva	
Nomeação	Regras gerais de nomeação File Names Type Names Variable Names Constant Names Section Names Namespace Names Enumerator Names Macro Names Exceptions to Naming Rules
Comentários	Comentário Style File Comentários Class Comments Function Comments Variable Comments Implementation Comments Punctuation, Spelling e Grammar TODO Comentários
Formatação	Comprimento da linha Não-ASCII Caracteres Spaces vs. Tabs Function Declarações e Definições Lambda Expressões SOrmitas Sas-ponto de partida Literals Function Chamadas De formatação de lista de inicialização debraced Ess Alás Sloops e declarações de switch Ponador e referência Expressões Boolean Expressões Revora valores Variáveis e de inicialização de array Desconstrução do processadores Class Format Constructor Initializer Lists Namespaces Formatting Horizontal Whitespace Vertical Whitespace
Exceções às Regras	Código de janelas não conforme existente

Fundo

C++ é um dos principais idiomas de desenvolvimento usados por muitos dos projetos de código aberto do Google. Como todo programador C++ sabe, a linguagem tem muitas características poderosas, mas esse poder traz consigo complexidade, o que, por sua vez, pode tornar o código mais propenso a bugs e mais difícil de ler e manter.

O objetivo deste guia é gerenciar essa complexidade descrevendo em detalhes os dos e não da escrita do código C++. Essas regras existem para manter a base de código gerenciável, permitindo que os codificadores usem recursos de

linguagem C++ de forma produtiva.

Estilo, também conhecido como legibilidade, é o que chamamos de convenções que regem nosso código C++. O termo Estilo é um pouco errado, já que essas convenções cobrem muito mais do que apenas a formatação de arquivos de origem.

A maioria dos projetos de código aberto desenvolvidos pelo Google estão de acordo com os requisitos deste guia.

Observe que este guia não é um tutorial C++: assumimos que o leitor está familiarizado com o idioma.

☞ Metas do Guia de Estilo

Por que temos esse documento?

Existem alguns objetivos fundamentais que acreditamos que este guia deve servir. Estes são os **motivos** fundamentais que fundamentam todas as regras individuais. Ao trazer essas ideias à frente, esperamos fundamentalizar discussões e deixar mais claro para nossa comunidade mais ampla por que as regras estão em vigor e por que decisões particulares foram tomadas. Se você entender quais objetivos cada regra está servindo, deve ser mais claro para todos quando uma regra pode ser dispensada (alguns podem ser), e que tipo de argumento ou alternativa seria necessário para mudar uma regra no guia.

Os objetivos do guia de estilo como os vemos atualmente são os seguintes:

Regras de estilo devem puxar seu peso

O benefício de uma regra de estilo deve ser grande o suficiente para justificar pedir a todos os nossos engenheiros que se lembrem dela. O benefício é medido em relação à base de código que obteríamos sem a regra, então uma regra contra uma prática muito prejudicial ainda pode ter um pequeno benefício se as pessoas não o fizerem de qualquer maneira. Esse princípio explica principalmente as regras que não temos, em vez das regras que temos: por exemplo, contraria muitos dos seguintes princípios, mas já é muito raro, de modo que o Guia de Estilo não discute isso.[goto](#)

Otimizar para o leitor, não para o escritor

Espera-se que nossa base de código (e a maioria dos componentes individuais submetidos a ela) continue por algum tempo. Como resultado, mais tempo será gasto lendo a maior parte do nosso código do que escrevê-lo. Optamos explicitamente por otimizar para a experiência de nosso código médio de leitura, manutenção e depuração de engenheiros de software em nossa base de código, em vez de facilitar ao escrever esse código. "Deixe um traço para o leitor" é um sub ponto particularmente comum deste princípio: Quando algo surpreendente ou incomum está acontecendo em um trecho de código (por exemplo, transferência de propriedade do ponteiro), deixar dicas textuais para o leitor no ponto de uso é valioso (demonstra a transferência de propriedade inequivocamente no site de chamadas). `std::unique_ptr`

Seja consistente com o código existente

Usar um estilo consistentemente através de nossa base de código nos permite focar em outras questões (mais importantes). A consistência também permite a automação: ferramentas que formatam seu código ou ajustam seu s só funcionam corretamente quando seu código é consistente com as expectativas da ferramenta. Em muitos casos, as regras que são atribuídas a "Ser Consistente" resumem-se a "Basta escolher um e parar de se preocupar com isso"; o valor potencial de permitir flexibilidade nesses pontos é superado pelo custo de ter pessoas discutindo sobre eles. No entanto, há limites para a consistência; é um bom desempate quando não há um argumento técnico claro, nem uma direção de longo prazo. Aplica-se mais fortemente localmente (por arquivo, ou para um conjunto de interfaces bem relacionadas). A consistência geralmente não deve ser usada como justificativa para fazer as coisas em um estilo antigo sem considerar os benefícios do novo estilo, ou a tendência da base de códigos convergir em estilos mais novos ao longo do tempo.[#include](#)

Seja consistente com a comunidade C++ mais ampla quando apropriado

A consistência com a forma como outras organizações usam C++ tem valor pelas mesmas razões que a consistência dentro da nossa base de código. Se um recurso no padrão C++ resolve um problema, ou se algum idioma é amplamente conhecido e aceito, esse é um argumento para usá-lo. No entanto, às vezes os recursos e expressões padrão são falhos, ou foram apenas projetados sem as necessidades da nossa base de código em mente. Nesses casos (como descrito abaixo) é apropriado restringir ou proibir recursos padrão. Em alguns casos, preferimos uma biblioteca caseira ou de terceiros em vez de uma biblioteca definida no Padrão C++, seja por superioridade percebida ou valor insuficiente para fazer a transição da base de código para a interface padrão.

Evite construções surpreendentes ou perigosas

C++ tem recursos que são mais surpreendentes ou perigosos do que se pode pensar de relance. Algumas restrições de guia de estilo estão em vigor para evitar cair nessas armadilhas. Há uma barra alta para renúncias de guia de estilo sobre tais restrições, pois renunciar a tais regras muitas vezes corre o risco de comprometer diretamente a correção do programa.

Evite construções que nosso programador C++ médio acharia complicado ou difícil de manter

C++ tem recursos que podem não ser geralmente apropriados devido à complexidade que eles introduzem ao código. Em código amplamente utilizado, pode ser mais aceitável usar construções linguísticas mais complicadas, pois quaisquer benefícios de implementação mais complexa são multiplicados amplamente pelo uso, e o custo em entender a complexidade não precisa ser pago novamente ao trabalhar com novas parcelas da base de código. Na dúvida, as renúncias a regras desse tipo podem ser procuradas perguntando aos leads do seu projeto. Isso é especificamente importante para nossa base de código porque a propriedade de códigos e a associação da equipe mudam ao longo do tempo: mesmo que todos que trabalham com algum código atualmente entendam, tal entendimento não é garantido para manter alguns anos a partir de agora.

Esteja atento à nossa escala

Com uma base de código de mais de 100 milhões de linhas e milhares de engenheiros, alguns erros e simplificações para um engenheiro podem se tornar caros para muitos. Por exemplo, é particularmente importante evitar poluir o namespace global: colisões de nomes em uma base de código de centenas de milhões de linhas são difíceis de trabalhar e difíceis de evitar se todos colocarem as coisas no namespace global.

Conceder a otimização quando necessário

As otimizações de desempenho às vezes podem ser necessárias e apropriadas, mesmo quando entram em conflito com os outros princípios deste documento.

A intenção deste documento é fornecer orientação máxima com restrição razoável. Como sempre, o bom senso e o bom gosto devem prevalecer. Por isso, referimos especificamente às convenções estabelecidas de toda a comunidade

Google C++, não apenas suas preferências pessoais ou as de sua equipe. Seja cético e relutante em usar construções inteligentes ou incomuns: a ausência de uma proibição não é o mesmo que uma licença para prosseguir. Use seu julgamento, e se você não tiver certeza, por favor, não hesite em pedir aos seus leads de projeto para obter informações adicionais.

↔ Versão C++

Atualmente, o código deve atingir C++17, ou seja, não deve usar recursos C++2x, com exceção dos [inicializadores designados](#). A versão C++ alvo deste guia avançará (agressivamente) ao longo do tempo.

Não utilize [extensões fora do padrão](#).

Considere a portabilidade para outros ambientes antes de usar recursos de C++14 e C++17 em seu projeto.

↔ Arquivos de cabeçalho

Em geral, cada arquivo deve ter um arquivo associado. Existem algumas exceções comuns, como testes unitários e pequenos arquivos contendo apenas uma função..cc.h.cemain()

O uso correto de arquivos de cabeçalho pode fazer uma enorme diferença para a legibilidade, tamanho e desempenho do seu código.

As seguintes regras irão guiá-lo através das várias armadilhas do uso de arquivos de cabeçalho.

↔ Cabeçalhos independentes

Os arquivos de cabeçalho devem ser independentes (compilar por conta própria) e terminar em . Os arquivos não-cabeçalho destinados à inclusão devem terminar e ser usados com moderação..h.inc

Todos os arquivos de cabeçalho devem ser independentes. Os usuários e as ferramentas de refatoração não devem aderir a condições especiais para incluir o cabeçalho. Especificamente, um cabeçalho deve ter [protetores de cabeçalho](#) e incluir todos os outros cabeçalhos que ele precisa.

Prefira colocar as definições para funções de modelo e inline no mesmo arquivo de suas declarações. As definições desses construtos devem ser incluídas em cada arquivo que os usa, ou o programa pode não vincular em algumas configurações de compilação. Se as declarações e definições estiverem em arquivos diferentes, incluindo o primeiro deve incluir transitivamente este último. Não mova essas definições para arquivos de cabeçalho incluídos separadamente (); essa prática era comum no passado, mas não é mais permitida..cc-inl.h

Como exceção, um modelo que é explicitamente instanciado para todos os conjuntos relevantes de argumentos de modelo, ou que é um detalhe de implementação privada de uma classe, é permitido ser definido no único arquivo que instancia o modelo..cc

Há casos raros em que um arquivo projetado para ser incluído não é independente. Estes são tipicamente destinados a ser incluídos em locais incomuns, como o meio de outro arquivo. Eles podem não usar [protetores de cabeçalho](#), e podem não incluir seus pré-requisitos. Nomeie esses arquivos com a extensão. Use com moderação e prefira cabeçalhos autônomos quando possível..inc

↔ A Guarda #define

Todos os arquivos de cabeçalho devem ter guardas para evitar a inclusão múltipla. O formato do nome do símbolo deve ser .#define<PROJECT>_<PATH>_<FILE>_H_

Para garantir a singularidade, eles devem ser baseados no caminho completo na árvore de origem de um projeto. Por exemplo, o arquivo no projeto deve ter o seguinte guarda:foo/src/bar/baz.hfoo

```
#ifndef FOO_BAR_BAZ_H_
#define FOO_BAR_BAZ_H_

...
#endif // FOO_BAR_BAZ_H_
```

↔ Inclua o que você usa

Se um arquivo de origem ou cabeçalho se referir a um símbolo definido em outro lugar, o arquivo deve incluir diretamente um arquivo de cabeçalho que pretende fornecer corretamente uma declaração ou definição desse símbolo. Não deve incluir arquivos de cabeçalho por qualquer outra razão.

Não conte com inclusões transitivas. Isso permite que as pessoas removam declarações não mais necessárias de seus cabeçalhos sem quebrar clientes. Isso também se aplica a cabeçalhos relacionados - deve incluir se ele usar um símbolo dele mesmo que inclua `#include foo.ccbar.hfoo.hbar.h`

↳ Declarações para a frente

Evite usar declarações para a frente sempre que possível. Em vez disso, [inclusões transitivas](#).

Definition:

Uma "declaração para a frente" é uma declaração de uma entidade sem uma definição associada.

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

Pros:

- As declarações de encaminhamento podem economizar tempo de compilação, pois forçam o compilador a abrir mais arquivos e processar mais entrada.`#include`
- As declarações para a frente podem economizar em recompilação desnecessária. Isso pode forçar seu código a ser recompilado com mais frequência, devido a alterações não relacionadas no cabeçalho.`#include`

Cons:

- As declarações de encaminhamento podem ocultar uma dependência, permitindo que o código do usuário pule a recompilação necessária quando os cabeçalhos mudam.
- Uma declaração para a frente em oposição a uma declaração `#include` dificulta a descoberta automática de ferramentas do módulo.
- Uma declaração para a frente pode ser quebrada por alterações subsequentes na biblioteca. As declarações de funções e modelos de encaminhamento podem impedir que os proprietários de cabeçalhos alterem suas APIs, como ampliar um tipo de parâmetro, adicionar um parâmetro de modelo com um valor padrão ou migrar para um novo namespace.
- A declaração de símbolos do namespace produz comportamento indefinido.`std::`
- Pode ser difícil determinar se uma declaração para a frente ou um completo é necessário. A substituição de um por uma declaração para a frente pode mudar silenciosamente o significado do código: Se o foi substituído por `decls` para frente e , chamaria `.#include#include`

```
// 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*)
```

- Para a frente, declarar vários símbolos de um cabeçalho pode ser mais verboso do que simplesmente incluir o cabeçalho.`#include`
- O código estruturante para habilitar declarações avançadas (por exemplo, usando membros de ponteiro em vez de membros do objeto) pode tornar o código mais lento e complexo.

Decision:

Tente evitar declarações antecipadas de entidades definidas em outro projeto.

↳ Funções inline

Defina funções inline somente quando forem pequenas, digamos, 10 linhas ou menos.

Definition:

Você pode declarar funções de uma maneira que permite ao compilador expandi-las em linha, em vez de chamá-las através do mecanismo usual de chamada de função.

Pros:

A inclinação de uma função pode gerar um código de objeto mais eficiente, desde que a função inline seja pequena. Sinta-se livre para inline acessórios e mutadores, e outras funções curtas e críticas de desempenho.

Cons:

O uso excessivo de inlining pode realmente tornar os programas mais lentos. Dependendo do tamanho de uma função, inlinando-a pode fazer com que o tamanho do código aumente ou diminua. A inulação de uma função de acessório muito pequena geralmente diminuirá o tamanho do código enquanto a inlina de uma função muito grande pode aumentar drasticamente o tamanho do código. Em processadores modernos, o código menor geralmente funciona mais rápido devido ao melhor uso do cache de instruções.

Decision:

Uma regra de ouro decente é não inline uma função se tiver mais de 10 linhas de comprimento. Cuidado com os destruidores, que muitas vezes são mais longos do que parecem por causa de chamadas implícitas de membros e destrutores de base!

Outra regra útil: normalmente não é rentável para funções inline com loops ou instruções de switch (a menos que, no caso comum, a instrução loop ou switch nunca seja executada).

É importante saber que as funções nem sempre são inlinadas, mesmo que sejam declaradas como tal; por exemplo, funções virtuais e recursivas normalmente não são inlinas. Normalmente, funções recursivas não devem ser inline. A principal razão para fazer uma função virtual inline é colocar sua definição na classe, seja por conveniência ou para documentar seu comportamento, por exemplo, para acessórios e mutadores.

» Nomes e Ordem de Inclusões

Inclua cabeçalhos na seguinte ordem: cabeçalho relacionado, cabeçalhos do sistema C, cabeçalhos de biblioteca padrão C++, cabeçalhos de outras bibliotecas, cabeçalhos do seu projeto.

Todos os arquivos de cabeçalho de um projeto devem ser listados como descendentes do diretório de origem do projeto sem o uso de codinomes de diretório UNIX (o diretório atual) ou (o diretório pai). Por exemplo, deve ser incluído como:...google-awesome-project/src/base/logging.h

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

Em ou , cujo principal objetivo é implementar ou testar o material em , ordenar o seu inclui o seguinte:*dir/foo.ccdir/foo_test.ccdir2/foo2.h*

1. *dir2/foo2.h*.
2. Uma linha em branco
3. Cabeçalhos do sistema C (mais precisamente: cabeçalhos em suportes angulares com a extensão), por exemplo, ..h<unistd.h><stdlib.h>
4. Uma linha em branco
5. Cabeçalhos de biblioteca padrão C++ (sem extensão de arquivo), por exemplo, <algorithm><cstddef>
6. Uma linha em branco
7. Os arquivos de outras bibliotecas..h
8. Uma linha em branco
9. Os arquivos do seu projeto..h

Separe cada grupo não vazio com uma linha em branco.

Com o pedido preferido, se o cabeçalho relacionado omitir qualquer necessidade inclui, a construção ou vai quebrar. Assim, essa regra garante que os intervalos de construção apareçam primeiro para as pessoas que trabalham nesses arquivos, não para pessoas inocentes em outros pacotes.*dir2/foo2.hdir/foo.ccdir/foo_test.cc*

dir/foo.cc e geralmente estão no mesmo diretório (por exemplo, e), mas às vezes podem estar em diretórios diferentes também.*dir2/foo2.hbase/basic_types_test.ccbase/basic_types.h*

Observe que os cabeçalhos C, como são essencialmente intercambiáveis com suas contrapartes C++(). Qualquer estilo é aceitável, mas prefira consistência com código existente.*stddef.hcstddef.hcstddef*

Dentro de cada seção, as inclui devem ser encomendadas alfabeticamente. Observe que o código mais antigo pode não estar em conformidade com esta regra e deve ser corrigido quando conveniente.

Por exemplo, as inclui podem ser assim:*google-awesome-project/src/foo/internal/fooserver.cc*

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

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

#include <string>
#include <vector>

#include "base/basic_types.h"
#include "base/commandlineflags.h"
#include "foo/server/bar.h"
```

Exceção:

Às vezes, o código específico do sistema precisa de itens condicionais. Esse código pode colocar inclui condicionada após outras inclui. Claro, mantenha seu código específico do sistema pequeno e localizado. Exemplo:

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

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

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

☞ Escopo

☞ Espaços de nome

Com poucas exceções, coloque o código em um namespace. Os namespaces devem ter nomes únicos com base no nome do projeto e, possivelmente, em seu caminho. Não use *diretivas de uso* (por exemplo, `#include <iostream>`). Não use espaços de nome inline. Para obter espaços de nome sem nome, consulte [Linkage Interno](#), using namespace foo

Definition:

Os namespaces subdividem o escopo global em escopos distintos e nomeados e, portanto, são úteis para evitar colisões de nomes no escopo global.

Pros:

Os namespaces fornecem um método para evitar conflitos de nomes em grandes programas, permitindo que a maioria dos códigos use nomes razoavelmente curtos.

Por exemplo, se dois projetos diferentes tiverem uma classe no escopo global, esses símbolos podem colidir no tempo de compilação ou no tempo de execução. Se cada projeto colocar seu código em um namespace, e agora são símbolos distintos que não colidem, e o código dentro do namespace de cada projeto pode continuar a se referir sem o prefixo. `Fooproject1::Fooproject2::FooFoo`

Os namespaces inline colocam automaticamente seus nomes no escopo de inc dentro. Considere o seguinte trecho, por exemplo:

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

As expressões são intercambiáveis. Os namespaces inline são destinados principalmente à compatibilidade de ABI entre as versões. `outer::inner::foo() outer::foo()`

Cons:

Os namespaces podem ser confusos, porque complicam a mecânica de descobrir a que definição um nome se refere.

Os namespaces inline, em particular, podem ser confusos porque os nomes não estão realmente restritos ao namespace onde eles são declarados. Eles só são úteis como parte de uma política de versionamento maior.

Em alguns contextos, é necessário referir-se repetidamente a símbolos por seus nomes totalmente qualificados. Para espaços de nomes profundamente aninhados, isso pode adicionar muita desordem.

Decision:

Os namespaces devem ser usados da seguinte forma:

- Siga as regras sobre [nomes do espaço de nome](#).
- Termine espaços de nome de várias linhas com comentários como mostrado nos exemplos apresentados.
- Os Namespaces envolvem todo o arquivo de origem após incluir, definições/declarações [gflags](#) e declarações de encaminhamento de classes de outros namespaces.

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

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

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

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

```

    ...
}

} // namespace mynamespace

```

Arquivos mais complexos podem ter detalhes adicionais, como bandeiras ou mensagens de uso..cc

```

#include "a.h"

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

namespace mynamespace {
    using ::foo::Bar;
    ...code for mynamespace...      // Code goes against the left margin.
} // namespace mynamespace

```

- Para colocar o código de mensagem de protocolo gerado em um namespace, use o especificador no arquivo. Consulte [pacotes de buffer de protocolo para obter detalhes](#).package.proto
- Não declare nada no namespace , incluindo declarações avançadas de aulas de biblioteca padrão. Declarar entidades no namespace é um comportamento indefinido, ou seja, não portável. Para declarar as entidades da biblioteca padrão, inclua o arquivo de cabeçalho apropriado.stdstd
- Você não pode usar uma *diretiva de uso* para disponibilizar todos os nomes de um namespace.

```

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

```

- Não use *codinomes* namespace no escopo namespace em arquivos de cabeçalho, exceto em namespaces explicitamente marcados apenas internamente, porque qualquer coisa importada em um namespace em um arquivo de cabeçalho torna-se parte da API pública exportada por esse arquivo.

```

// 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
}

```

- Não use espaços de nome inline.

☞ Ligação Interna

Quando as definições em um arquivo não precisarem ser referenciadas fora desse arquivo, dê-lhes vinculação interna colocando-as em um namespace não nomeado ou declarando-as . Não use nenhuma dessas construções em arquivos .ccstatic.h

Definition:

Todas as declarações podem ser dadas vinculação interna colocando-as em namespaces não nomeados. Funções e variáveis também podem ser dadas vinculação interna declarando-as . Isso significa que qualquer coisa que você está declarando não pode ser acessado de outro arquivo. Se um arquivo diferente declara algo com o mesmo nome, então as duas entidades são completamente independentes.static

Decision:

O uso de linkage interno em arquivos é incentivado para todos os códigos que não precisam ser referenciados em outros lugares. Não use linkage interno em arquivos..cc.h

Formatar espaços de nome sem nome como namespaces chamados. No comentário final, deixe o nome do namespace vazio:

```

namespace {
...
} // namespace

```

⇒ Funções não-membros, estáticas e globais

Prefira colocar funções não-membros em um namespace; usar funções completamente globais raramente. Não use uma classe simplesmente para agrupar membros estáticos. Os métodos estáticos de uma classe devem geralmente estar intimamente relacionados com instâncias da classe ou dados estáticos da classe.

Pros:

Funções de membros não membros e estáticos podem ser úteis em algumas situações. Colocar funções não-membros em um namespace evita poluir o namespace global.

Cons:

Funções de membros não membros e estáticos podem fazer mais sentido como membros de uma nova classe, especialmente se acessarem recursos externos ou tiverem dependências significativas.

Decision:

Às vezes é útil definir uma função não vinculada a uma instância de classe. Tal função pode ser um membro estático ou uma função não-membro. As funções não-membros não devem depender de variáveis externas, e devem quase sempre existir em um namespace. Não crie classes apenas para membros estáticos de grupo; isso não é diferente de apenas dar aos nomes um prefixo comum, e esse agrupamento geralmente é desnecessário de qualquer maneira.

Se você definir uma função não-membro e ela só for necessária em seu arquivo, use [a vinculação interna](#) para limitar seu escopo..cc

⇒ Variáveis Locais

Coloque as variáveis de uma função no escopo mais estreito possível e inicialize variáveis na declaração.

C++ permite que você declare variáveis em qualquer lugar em uma função. Nós encorajamos você a declará-los no mais local possível, e o mais próximo possível do primeiro uso possível. Isso facilita para o leitor encontrar a declaração e ver qual é o tipo da variável e para que ela foi inicializada. Em particular, a inicialização deve ser usada em vez de declaração e atribuição, por exemplo,:

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

```
int j = g(); // Good -- declaration has initialization.
```

```
std::vector<int> v;  
v.push_back(1); // Prefer initializing using brace initialization.  
v.push_back(2);
```

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

Variáveis necessárias , e declarações devem ser normalmente declaradas dentro dessas declarações, de modo que tais variáveis se limitam a esses escopos. Por exemplo:ifwhilefor

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

Há uma ressalva: se a variável é um objeto, seu construtor é invocado toda vez que entra no escopo e é criado, e seu destruidor é invocado toda vez que sai do escopo.

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

Pode ser mais eficiente declarar tal variável usada em um loop fora desse loop:

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

⇒ Variáveis Estáticas e Globais

Objetos com [duração de armazenamento estático](#) são proibidos a menos que sejam [trivialmente destrutíveis](#). Informalmente, isso significa que o destruidor não faz nada, mesmo levando em conta os destrutores de membros e

bases. Mais formalmente, significa que o tipo não tem um destrutor virtual ou definido pelo usuário e que todas as bases e membros não estáticos são trivialmente destrutíveis. Variáveis locais de função estática podem usar inicialização dinâmica. O uso da inicialização dinâmica para variáveis ou variáveis de membros de classe estática no escopo do namespace é desencorajado, mas permitido em circunstâncias limitadas; veja abaixo para mais detalhes.

Como regra geral: uma variável global satisfaz esses requisitos se sua declaração, considerada isoladamente, poderia ser `.constexpr`.

Definition:

Cada objeto tem uma *duração de armazenamento*, que se correlaciona com sua vida útil. Objetos com duração de armazenamento estático ao vivo desde o ponto de sua inicialização até o final do programa. Tais objetos aparecem como variáveis no escopo do namespace ("variáveis globais"), como membros de dados estáticos das classes, ou como variáveis locais de função que são declaradas com o especificador. As variáveis estáticas locais da função são iniciadas quando o controle passa pela primeira vez através de sua declaração; todos os outros objetos com duração de armazenamento estático são iniciados como parte da inicialização do programa. Todos os objetos com duração de armazenamento estático são destruídos na saída do programa (o que acontece antes que os fios não adjacentes sejam terminados).`static`

A inicialização pode ser *dinâmica*, o que significa que algo não trivial acontece durante a inicialização. (Por exemplo, considere um construtor que aloca a memória ou uma variável inicializada com o ID de processo atual.) O outro tipo de inicialização é a *inicialização estática*. Os dois não são completamente opostos, porém: a inicialização estática *sempre* acontece com objetos com duração de armazenamento estático (inicializando o objeto seja a uma determinada constante ou a uma representação composta por todos os bytes definidos a zero), enquanto a inicialização dinâmica acontece depois disso, se necessário.

Pros:

Variáveis globais e estáticas são muito úteis para um grande número de aplicações: constantes nomeadas, estruturas de dados auxiliares internas para alguma unidade de tradução, bandeiras de linha de comando, registro, mecanismos de registro, infraestrutura de fundo, etc.

Cons:

Variáveis globais e estáticas que usam inicialização dinâmica ou possuem destrutores não triviais criam complexidade que pode facilmente levar a bugs difíceis de encontrar. A inicialização dinâmica não é ordenada entre unidades de tradução, e nem a destruição (exceto que a destruição acontece em ordem inversa de inicialização). Quando uma inicialização se refere a outra variável com duração de armazenamento estático, é possível que isso faz com que um objeto seja acessado antes de sua vida útil ter começado (ou após o término de sua vida útil). Além disso, quando um programa inicia threads que não são aderidos na saída, esses threads podem tentar acessar objetos após o término de sua vida se seu destruidor já tiver sido executado.

Decision:

Decisão sobre destruição

Quando os destruidores são triviais, sua execução não está sujeita a ordenar em tudo (eles efetivamente não são "executados"); caso contrário, estamos expostos ao risco de acessar objetos após o fim de sua vida. Portanto, só permitimos objetos com duração de armazenamento estático se forem trivialmente destrutíveis. Tipos fundamentais (como ponteiros e) são trivialmente destrutíveis, assim como matrizes de tipos trivialmente destrutíveis. Observe que as variáveis marcadas são trivialmente destrutíveis.`.intconstexpr`

```
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}};
}
```

Observe que as referências não são objetos e, portanto, não estão sujeitas às restrições à destrusibilidade. A restrição da inicialização dinâmica ainda se aplica, no entanto. Em particular, é permitida uma referência estática local da função do formulário.`static T& t = *new T;`

Decisão sobre inicialização

A inicialização é um tema mais complexo. Isso porque não devemos considerar apenas se os construtores de classe executam, mas também devemos considerar a avaliação do inicializador:

```
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)
```

Todos, exceto a primeira declaração, nos expõem a pedidos de inicialização indeterminados.

O conceito que estamos procurando é chamado de *inicialização constante* na linguagem formal do padrão C++. Significa que a expressão inicializante é uma expressão constante, e se o objeto é inicializado por uma chamada de construtor, então o construtor deve ser especificado como, também:`constexpr`

```
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 constexpr
Foo a[] = { Foo(1), Foo(2), Foo(3) }; // fine
```

A inicialização constante é sempre permitida. A inicialização constante das variáveis de duração do armazenamento estático deve ser marcada com ou, sempre que possível, o atributo `ABSL_CONST_INIT`. Qualquer variável de duração de armazenamento estático não local que não esteja tão marcada deve ser presumidamente com inicialização dinâmica e revisada com muito cuidado.`constexpr`

Em contrapartida, as seguintes inicializações são problemáticas:

```
// 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
```

A inicialização dinâmica de variáveis não locais é desencorajada e, em geral, é proibida. No entanto, permitimos que nenhum aspecto do programa dependa do sequenciamento dessa inicialização em relação a todas as outras inicializações. Sob essas restrições, o ordenamento da inicialização não faz diferença observável. Por exemplo:

```
int p = getpid(); // allowed, as long as no other static variable
                  // uses p in its own initialization
```

A inicialização dinâmica das variáveis locais estáticas é permitida (e comum).

Padrões comuns

- Cordas globais: se você precisar de uma constante de string global ou estática nomeada, considere usar uma variável de , matriz de caracteres ou ponteiro de caractere, apontando para uma sequência literal. Os literais de cordas já têm duração estática de armazenamento e geralmente são suficientes. Veja [TotW #140](#),`constexprstring_view`
- Mapas, conjuntos e outros recipientes dinâmicos: se você precisar de uma coleção estática e fixa, como um conjunto para pesquisar contra ou uma tabela de pesquisa, você não pode usar os recipientes dinâmicos da biblioteca padrão como uma variável estática, uma vez que eles têm destrutores não triviais. Em vez disso, considere uma simples matriz de tipos triviais, por exemplo, uma matriz de matrizes de ints (para um "mapa do int ao int"), ou uma matriz de pares (por exemplo, pares de e). Para pequenas coleções, a busca linear é inteiramente suficiente (é eficiente, devido à localidade de memória); considere usar as instalações de [absl/algoritmo/container.h](#) para as operações padrão. Se necessário, mantenha a coleta em ordem classificada e use um algoritmo de busca binário. Se você realmente preferir um recipiente dinâmico da biblioteca padrão, considere usar um ponteiro estático local de função, conforme descrito abaixo.`int const char*`
- Ponteiros inteligentes (): ponteiros inteligentes executam a limpeza durante a destruição e, portanto, são proibidos. Considere se seu caso de uso se encaixa em um dos outros padrões descritos nesta seção. Uma solução simples é usar um ponteiro simples em um objeto alocado dinamicamente e nunca excluí-lo (ver o último item).`unique_ptr/shared_ptr`
- Variáveis estáticas de tipos personalizados: se você precisar de dados estáticos e constantes de um tipo que você precisa definir a si mesmo, dê ao tipo um destruidor trivial e um construtor.`constexpr`
- Se tudo falhar, você pode criar um objeto dinamicamente e nunca excluí-lo usando um ponteiro ou referência estática local (por exemplo, `static const auto& impl = *new T(args...);`)

☞ Variáveis `thread_local`

`thread_local` variáveis que não são declaradas dentro de uma função devem ser inicializadas com uma verdadeira constante de tempo de compilação, e isso deve ser aplicado usando o atributo `ABSL_CONST_INIT`. Prefira outras formas de definir dados locais de segmento.`thread_local`

Definition:

As variáveis podem ser declaradas com o especificador:`thread_local`

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

Tal variável é na verdade uma coleção de objetos, de modo que quando diferentes segmentos acessam, eles estão realmente acessando diferentes objetos, as variáveis são muito [parecidas com variáveis de duração de armazenamento estático](#) em muitos aspectos. Por exemplo, elas podem ser declaradas no escopo do namespace, em funções internas ou como membros de classe estática, mas não como membros comuns da classe.`thread_local`

`thread_local` instâncias variáveis são inicializadas muito como variáveis estáticas, exceto que elas devem ser inicializadas separadamente para cada segmento, em vez de uma vez na inicialização do programa. Isso significa que as variáveis declaradas dentro de uma função são seguras, mas outras variáveis estão sujeitas aos mesmos problemas de ordem de inicialização que as variáveis estáticas (e muito mais).`thread_local`

`thread_local` instâncias variáveis não são destruídas antes que seu segmento termine, de modo que elas não têm os problemas de ordem de destruição de variáveis estáticas.

Pros:

- Os dados locais de rosca são inherentemente seguros contra raças (porque apenas um segmento pode acessá-los normalmente), o que torna útil para a programação simultânea.`thread_local`
- `thread_local` é a única maneira suportada por padrões de criar dados locais de segmento.

Cons:

- Acessar uma variável pode desencadear a execução de uma quantidade imprevisível e incontrolável de outro código.`thread_local`
- `thread_local` as variáveis são efetivamente variáveis globais, e têm todas as desvantagens de variáveis globais que não a falta de segurança de rosca.
- A memória consumida por uma balança variável com o número de rosas em execução (na pior das hipóteses), que podem ser bastante grandes em um programa.`thread_local`
- Os membros de dados não estáticos não podem ser `.thread_local`
- `thread_local` pode não ser tão eficiente quanto certos intrínsecos do compilador.

Decision:

`thread_local` variáveis dentro de uma função não têm preocupações de segurança, para que possam ser usadas sem restrições. Observe que você pode usar um escopo de função para simular um escopo de classe ou namespace definindo uma função ou método estático que o exponha:`thread_local`

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

`thread_local` as variáveis no escopo de classe ou namespace devem ser inicializadas com uma verdadeira constante de tempo de compilação (ou seja, elas não devem ter inicialização dinâmica). Para fazer isso, as variáveis no escopo de classe ou namespace devem ser anotadas com [ABSL_CONST_INIT](#) (ou , mas isso deve ser raro):`thread_local`

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

`thread_local` deve ser preferido em relação a outros mecanismos para definir dados locais de segmento.

Classes

As aulas são a unidade fundamental do código em C++. Naturalmente, nós os usamos extensivamente. Esta seção lista os principais dos e não que você deve seguir ao escrever uma aula.

Fazendo trabalhos em construtores

Evite chamadas de método virtual em construtores e evite a inicialização que pode falhar se você não puder sinalizar um erro.

Definition:

É possível realizar a inicialização arbitrária no corpo do construtor.

Pros:

- Não há necessidade de se preocupar se a classe foi iniciada ou não.
- Objetos que são totalmente iniciados por chamada de construtor podem ser e também podem ser mais fáceis de usar com recipientes ou algoritmos padrão.`const`

Cons:

- Se o trabalho chamar funções virtuais, essas chamadas não serão enviadas para as implementações da subclasse. A modificação futura na sua classe pode introduzir esse problema discretamente mesmo que sua classe não esteja subclassificada no momento, causando muita confusão.
- Não há uma maneira fácil de os construtores sinalizarem erros, a não ser travar o programa (nem sempre apropriado) ou usar exceções (que são [proibidas](#)).
- Se o trabalho falhar, agora temos um objeto cujo código de inicialização falhou, então pode ser um estado incomum exigindo um mecanismo de verificação de estado (ou semelhante) que é fácil de esquecer de chamar.`bool IsValid()`
- Você não pode pegar o endereço de um construtor, então qualquer trabalho feito no construtor não pode ser facilmente entregue a, por exemplo, outro segmento.

Decision:

Os construtores nunca devem chamar funções virtuais. Se apropriado para o seu código, encerrar o programa pode ser uma resposta adequada para o tratamento de erros. Caso contrário, considere uma função de fábrica ou método como descrito no [TotW #42](#). Evite métodos em objetos sem outros estados que afetem quais métodos públicos podem ser chamados (objetos semiconstídos desta forma são particularmente difíceis de trabalhar corretamente).`Init()`

☞ Conversões Implícitas

Não defina conversões implícitas. Use a palavra-chave para operadores de conversão e construtores de argumento único.`explicit`

Definition:

Conversões implícitas permitem que um objeto de um tipo (chamado de *tipo de origem*) seja usado onde um tipo diferente (chamado de *tipo de destino*) é esperado, como ao passar um argumento para uma função que toma um parâmetro.`int double`

Além das conversões implícitas definidas pelo idioma, os usuários podem definir os seus próprios, adicionando membros apropriados à definição de classe do tipo de origem ou destino. Uma conversão implícita no tipo de origem é definida por um operador de conversão de tipo nomeado após o tipo de destino (por exemplo, `operator int()`). Uma conversão implícita no tipo de destino é definida por um construtor que pode tomar o tipo de origem como seu único argumento (ou apenas argumento sem valor padrão)`operator bool()`

A palavra-chave pode ser aplicada a um construtor ou a um operador de conversão, para garantir que ele só possa ser usado quando o tipo de destino estiver explícito no ponto de uso, por exemplo, com um molde. Isso se aplica não apenas a conversões implícitas, mas a listar a sintaxe de inicialização:`explicit`

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

void Func(Foo f);
```

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

Este tipo de código não é tecnicamente uma conversão implícita, mas a linguagem o trata como um no que diz respeito a `explicit`

Pros:

- Conversões implícitas podem tornar um tipo mais utilizável e expressivo eliminando a necessidade de nomear explicitamente um tipo quando é óbvio.
- Conversões implícitas podem ser uma alternativa mais simples à sobrecarga, como quando uma única função com um parâmetro toma o lugar de sobrecargas separadas para `e .string_view std::string const char*`
- A sintaxe de inicialização da lista é uma forma concisa e expressiva de inicializar objetos.

Cons:

- Conversões implícitas podem ocultar bugs de incompatibilidade de tipo, onde o tipo de destino não corresponde à expectativa do usuário, ou o usuário não sabe que qualquer conversão ocorrerá.
- Conversões implícitas podem tornar o código mais difícil de ler, particularmente na presença de sobrecarga, tornando-o menos óbvio qual código está realmente sendo chamado.
- Os construtores que tomam um único argumento podem accidentalmente ser utilizáveis como conversões implícitas do tipo, mesmo que não se destinem a fazê-lo.
- Quando um construtor de argumento único não está marcado, não há uma maneira confiável de dizer se ele pretende definir uma conversão implícita, ou o autor simplesmente esqueceu de marcá-lo.`explicit`
- Conversões implícitas podem levar a ambiguidades de call-site, especialmente quando há conversões implícitas bidirecionais. Isso pode ser causado por ter dois tipos que ambos fornecem uma conversão implícita, ou por um único tipo que tem um construtor implícito e um operador de conversão de tipo implícito.
- A inicialização da lista pode sofrer com os mesmos problemas se o tipo de destino estiver implícito, especialmente se a lista tiver apenas um único elemento.

Decision:

Os operadores de conversão de tipos e os construtores que são callable com um único argumento devem ser marcados na definição de classe. Como exceção, os construtores copiadores e de movimento não devem ser, uma vez que não realizam conversão de tipo.`explicit`

Conversões implícitas às vezes podem ser necessárias e apropriadas para tipos que são projetados para serem intercambiáveis, por exemplo, quando objetos de dois tipos são apenas representações diferentes do mesmo valor

subjacente. Nesse caso, entrar em contato com seu projeto leva a solicitar uma renúncia desta regra.

Construtores que não podem ser chamados com um único argumento podem omitir . Os construtores que tomam um único parâmetro também devem omitir, a fim de suportar a inicialização da cópia (por exemplo,).
).explicitstd::initializer_list<MyType> m = {1, 2};

↔️ Tipos copiáveis e móveis

A API pública de uma classe deve deixar claro se a classe é copiável, somente em movimento, ou não copiável nem móvel. Apoie copiar e/ou mover se essas operações forem claras e significativas para o seu tipo.

Definition:

Um tipo móvel é aquele que pode ser inicializado e atribuído a partir de temporários.

Um tipo copiável é aquele que pode ser inicializado ou atribuído a qualquer outro objeto do mesmo tipo (por isso também é móvel por definição), com a estipulação de que o valor da fonte não muda, é um exemplo de um tipo móvel, mas não copiável (uma vez que o valor da fonte deve ser modificado durante a cessão ao destino). e são exemplos de tipos móveis que também são copiáveis. (Para , as operações de movimentação e cópia são as mesmas; para , existe uma operação de movimento que é menos cara do que uma cópia.)
std::unique_ptr<int> std::unique_ptr<int> int std::string int std::string

Para tipos definidos pelo usuário, o comportamento da cópia é definido pelo construtor de cópias e pelo operador de atribuição de cópia. O comportamento de movimento é definido pelo construtor de movimento e pelo operador de atribuição de movimento, se eles existirem, ou pelo construtor de cópias e pelo operador de atribuição de cópia de outra forma.

Os construtores de cópia/movimento podem ser implicitamente invocados pelo compilador em algumas situações, por exemplo, ao passar objetos por valor.

Pros:

Objetos de tipos copiáveis e móveis podem ser passados e devolvidos por valor, o que torna as APIs mais simples, seguras e mais gerais. Ao contrário de quando se passa objetos por ponteiro ou referência, não há risco de confusão sobre propriedade, vida, mutabilidade e questões semelhantes, e não há necessidade de especificá-los no contrato. Também evita interações não locais entre o cliente e a implementação, o que os torna mais fáceis de entender, manter e otimizar pelo compilador. Além disso, esses objetos podem ser usados com APIs genéricas que requerem valor de passagem, como a maioria dos contêineres, e permitem flexibilidade adicional na composição do tipo, por exemplo.

Construtores de cópia/movimentação e operadores de atribuição geralmente são mais fáceis de definir corretamente do que alternativas como , ou , porque eles podem ser gerados pelo compilador, implícita ou com . Eles são conciso e garantem que todos os membros de dados sejam copiados. Os construtores de cópias e movimentos também são geralmente mais eficientes, porque eles não exigem alocação de montes ou etapas de inicialização e atribuição separadas, e eles são elegíveis para otimizações, como [elision de cópia](#).Clone()CopyFrom()Swap()= default

As operações de movimentação permitem a transferência implícita e eficiente de recursos de objetos de rvalue. Isso permite um estilo de codificação mais simples em alguns casos.

Cons:

Alguns tipos não precisam ser copiáveis, e fornecer operações de cópia para esses tipos pode ser confuso, sem sentido ou totalmente incorreto. Os tipos que representam objetos singleton (), objetos ligados a um escopo específico (), ou intimamente acoplados à identidade do objeto () não podem ser copiados significativamente. As operações de cópia para tipos de classe base que devem ser usadas polimorficamente são perigosas, porque o uso deles pode levar ao [corte de objetos](#). As operações de cópia padrão ou implementadas descuidadamente podem ser incorretas, e os bugs resultantes podem ser confusos e difíceis de diagnosticar.
RegistererCleanupMutex

Construtores de cópias são invocados implicitamente, o que torna a invocação fácil de perder. Isso pode causar confusão para programadores acostumados a linguagens onde o passe por referência é convencional ou obrigatório. Também pode incentivar a cópia excessiva, o que pode causar problemas de desempenho.

Decision:

A interface pública de cada classe deve deixar claro qual copiar e mover as operações que a classe suporta. Isso geralmente deve assumir a forma de declarar e/ou excluir explicitamente as operações apropriadas na seção da declaração.`public`

Especificamente, uma classe copiável deve declarar explicitamente as operações de cópia, uma classe somente para movimentos deve declarar explicitamente as operações de movimento e uma classe não copiável/móvel deve excluir explicitamente as operações de cópia. Uma classe copiável também pode declarar operações de movimento para suportar movimentos eficientes. Declarar ou excluir explicitamente todas as quatro operações de cópia/movimentação é permitido, mas não é necessário. Se você fornecer uma cópia ou mover o operador de atribuição, você também deve fornecer o construtor correspondente.

```
class Copyable {
public:
    Copyable(const Copyable& other) = default;
    Copyable& operator=(const Copyable& other) = default;

    // The implicit move operations are suppressed by the declarations above.
    // You may explicitly declare move operations to support efficient moves.
};

class MoveOnly {
public:
    MoveOnly(const MoveOnly& other) = default;
```

```

moveonly(moveonly& other) = default;
MoveOnly& operator=(MoveOnly&& other) = default;

// The copy operations are implicitly deleted, but you can
// spell that out explicitly if you want:
MoveOnly(const MoveOnly&) = delete;
MoveOnly& operator=(const MoveOnly&) = delete;
};

class NotCopyableOrMovable {
public:
    // Not copyable or movable
    NotCopyableOrMovable(const NotCopyableOrMovable&) = delete;
    NotCopyableOrMovable& operator=(const NotCopyableOrMovable&)
        = delete;

    // The move operations are implicitly disabled, but you can
    // spell that out explicitly if you want:
    NotCopyableOrMovable(NotCopyableOrMovable&&) = delete;
    NotCopyableOrMovable& operator=(NotCopyableOrMovable&&)
        = delete;
};

```

Essas declarações/exclusões só podem ser omitidas se forem óbvias:

- Se a classe não tiver seção, como uma [estrutura](#) ou uma classe base somente de interface, então a cópia/mobilidadeabilidade pode ser determinada pela cópia/mobilidadeabilidade de qualquer membro de dados públicos. `private`
- Se uma classe base claramente não é copiável ou móvel, as classes derivadas naturalmente também não serão. Uma classe base somente de interface que deixa essas operações implícitas não é suficiente para deixar as subclasses concretas claras.
- Observe que se você declarar ou excluir explicitamente a operação de construtor ou cessão para cópia, a outra operação de cópia não é óbvia e deve ser declarada ou excluída. Da mesma forma para operações de mudança.

Um tipo não deve ser copiável/móvel se o significado de copiar/mover não é claro para um usuário casual ou se ele incorre em custos inesperados. As operações de movimento para tipos copiáveis são estritamente uma otimização de desempenho e são uma fonte potencial de bugs e complexidade, por isso evite defini-los a menos que sejam significativamente mais eficientes do que as operações de cópia correspondentes. Se o seu tipo fornecer operações de cópia, é recomendável que você projete sua classe para que a implementação padrão dessas operações esteja correta. Lembre-se de revisar a correção de quaisquer operações padrão como você faria com qualquer outro código.

Para eliminar o risco de fatiamento, prefira fazer as classes básicas abstratas, tornando seus construtores protegidos, declarando seus destrutores protegidos, ou dando-lhes uma ou mais funções de membros virtuais puros. Prefira evitar o desocremento de classes concretas.

☞ Estruturas vs. Classes

Use um somente para objetos passivos que transportam dados; todo o resto é um `.struct``.class`

As palavras-chave se comportam quase de forma idêntica em C++. Adicionamos nossos próprios significados semânticos a cada palavra-chave, então você deve usar a palavra-chave apropriada para o tipo de dados que você está definindo.`.struct``.class`

`structs` devem ser usados para objetos passivos que carregam dados, e podem ter constantes associadas. Todos os campos devem ser públicos. A estrutura não deve ter invariantes que impliquem relações entre diferentes campos, uma vez que o acesso direto do usuário a esses campos pode quebrar essas invariantes. Podem estar presentes construtores, destruidores e métodos de ajuda; no entanto, esses métodos não devem exigir ou impor quaisquer invariantes.

Se forem necessárias mais funcionalidades ou invariantes, a é mais apropriada. Em caso de dúvida, faça disso `um.class``.class`

Para obter consistência com STL, você pode usar em vez de para tipos apátridas, como traços, [metafunções de modelo](#) e alguns funtores.`.struct``.class`

Observe que as variáveis membros em estruturas e classes têm [regras de nomeação diferentes](#).

☞ Structs vs. Pares e Tuplas

Prefira usar um em vez de um par ou uma tupla sempre que os elementos podem ter nomes significativos.`.struct`

Embora o uso de pares e tuplas possa evitar a necessidade de definir um tipo personalizado, potencialmente salvando o trabalho ao escrever código, um nome de campo significativo será quase sempre muito mais claro ao ler código do que , ou . Enquanto a introdução do C++14 de acessar um elemento tupla por tipo em vez de índice (quando o tipo é único) pode às vezes mitigar parcialmente isso, um nome de campo geralmente é substancialmente mais claro e informativo do que um tipo. `.first.second``std::get<Type>`

Pares e tuplas podem ser apropriados em código genérico onde não há significados específicos para os elementos do par ou tupla. Seu uso também pode ser necessário para interagir com código ou APIs existentes.

☞ Herança

A composição é muitas vezes mais apropriada do que a herança. Ao usar a herança, faça-a.public

Definition:

Quando uma sub-classe herda de uma classe base, ela inclui as definições de todos os dados e operações que a classe base define. "Herança de interface" é herança de uma classe base abstrata pura (uma sem estado ou métodos definidos); todas as outras heranças são "herança de implementação".

Pros:

A herança de implementação reduz o tamanho do código reutilizando o código de classe base, pois ele se especializa com um tipo existente. Como a herança é uma declaração de tempo de compilação, você e o compilador podem entender a operação e detectar erros. A herança da interface pode ser usada para impor programáticamente que uma classe exponha uma API específica. Novamente, o compilador pode detectar erros, neste caso, quando uma classe não define um método necessário da API.

Cons:

Para a herança de implementação, como o código que implementa uma sub-classe está espalhado entre a base e a sub-classe, pode ser mais difícil entender uma implementação. A sub-classe não pode substituir funções que não são virtuais, portanto, a sub-classe não pode alterar a implementação.

A herança múltipla é especialmente problemática, porque muitas vezes impõe uma sobrecarga de desempenho mais alta (na verdade, a queda de desempenho da herança única para a herança múltipla pode muitas vezes ser maior do que a queda de desempenho do despacho comum para o virtual), e porque corre o risco de levar a padrões de herança "diamante", que são propensos a ambiguidade, confusão e erros absolutos.

Decision:

Toda herança deve ser. Se você quer fazer herança privada, você deve estar incluindo uma instância da classe base como membro em vez disso. Você pode usar em aulas quando não pretende apoiar usá-las como classes base.publicfinal

Não use demais a herança de implementação. A composição é muitas vezes mais apropriada. Tente restringir o uso da herança para o caso "is-a": subclasses se pode razoavelmente ser dito que "é uma espécie de".BarFooBarFoo

Limitar o uso dessas funções membros que podem precisar ser acessadas a partir de subclasses. Observe que [os membros dos dados devem ser privados](#).protected

Anote explicitamente substituições de funções virtuais ou destrutores virtuais com exatamente um de um especificador ou (menos frequentemente). Não use ao declarar uma substituição. Racionalidade: Uma função ou destruidor marcado ou que não é uma substituição de uma função virtual de classe base não será compilado, e isso ajuda a pegar erros comuns. Os especificadores servem como documentação; se não houver especificador, o leitor tem que verificar todos os ancestrais da classe em questão para determinar se a função ou destruidor é virtual ou não.overridefinalvirtualoverridefinal

Uma herança múltipla é permitida, mas a herança de *implementação* múltipla é fortemente desencorajada.

☞ Sobrecarga do operador

Sobrecarga dos operadores criteriosamente. Não use literais definidos pelo usuário.

Definition:

O C++ permite que o código [do usuário declare versões sobrecarregadas dos operadores incorporados usando a palavra-chave](#), desde que um dos parâmetros seja um tipo definido pelo usuário. A palavra-chave também permite que o código do usuário defina novos tipos de literais usando e defina funções de conversão de tipo, tais como .operatoroperatoroperator""operator bool()

Pros:

A sobrecarga do operador pode tornar o código mais conciso e intuitivo, permitindo que os tipos definidos pelo usuário se comportem da mesma forma que os tipos incorporados. Operadores sobrecarregados são os nomes idiomáticos para determinadas operações (por exemplo, , e), e aderir a essas convenções pode tornar os tipos definidos pelo usuário mais legíveis e permitir que eles interoperem com bibliotecas que esperam esses nomes.==<=<<

Os literais definidos pelo usuário são uma notação muito concisa para criar objetos de tipos definidos pelo usuário.

Cons:

- Fornecer um conjunto correto, consistente e surpreendente de sobrecargas de operadores requer alguns cuidados, e não fazê-lo pode levar a confusão e bugs.
- O uso excessivo de operadores pode levar a código ofuscado, especialmente se a semântica do operador sobrecarregado não seguir a convenção.
- Os riscos de sobrecarga de função aplicam-se tanto à sobrecarga do operador, se não mais.
- As sobrecargas dos operadores podem enganar nossa intuição em pensar que operações caras são operações baratas e incorporadas.
- Encontrar os sites de chamada para operadores sobrecarregados pode exigir uma ferramenta de pesquisa que esteja ciente da sintaxe C++, em vez de, por exemplo, grep.

- Se você errar o tipo de argumento de um operador sobre carregado, você pode obter uma sobre carga diferente do que um erro do compilador. Por exemplo, pode fazer uma coisa, enquanto faz algo totalmente diferente.`foo < bar&foo < &bar`
- Certas sobre cargas de operadores são inherentemente perigosas. A sobre carga pode fazer com que o mesmo código tenha significados diferentes, dependendo se a declaração de sobre carga é visível. Sobre cargas de `,`, `e` (`círculo`) não podem coincidir com a semântica de ordem de avaliação dos operadores incorporados.`&&||`,
- Os operadores são muitas vezes definidos fora da classe, então há o risco de diferentes arquivos introduzirem definições diferentes do mesmo operador. Se ambas as definições estiverem ligadas ao mesmo binário, isso resulta em comportamento indefinido, que pode se manifestar como bugs sutis de tempo de execução.
- Os literais definidos pelo usuário (UDLs) permitem a criação de novas formas sintáticas que não são familiares até mesmo para programadores C++ experientes, como uma abreviação para `.`. As notações existentes são mais claras, embora menos terse.`"Hello World"svstd::string_view("Hello World")`
- Como eles não podem ser qualificados para o namespace, os usos de UDLs também exigem o uso de diretivas de uso (que [proibimos](#)) ou o uso de declarações (que [proibimos em arquivos de cabeçalho](#), exceto quando os nomes importados fazem parte da interface exposta pelo arquivo de cabeçalho em questão). Dado que os arquivos de cabeçalho teriam que evitar sufixos UDL, preferimos evitar que convenções para literais diferem entre arquivos de cabeçalho e arquivos de origem.

Decision:

Definir operadores sobre carregados somente se seu significado for óbvio, surpreendente e consistente com os operadores incorporados correspondentes. Por exemplo, use como um bitwise ou lógico- ou, não como um tubo estilo shell.`|`

Defina os operadores apenas em seus próprios tipos. Mais precisamente, defina-os nos mesmos cabeçalhos, arquivos `.cc` e namespaces como os tipos em que operam. Dessa forma, os operadores estão disponíveis onde quer que o tipo esteja, minimizando o risco de múltiplas definições. Se possível, evite definir os operadores como modelos, pois eles devem satisfazer essa regra para quaisquer possíveis argumentos de modelo. Se você definir um operador, também defina quaisquer operadores relacionados que façam sentido e certifique-se de que eles sejam definidos de forma consistente. Por exemplo, se você sobre carregar, sobre carregar todos os operadores de comparação e certifique-se de e nunca retornar verdadeiro para os mesmos argumentos.<<>

Prefira definir operadores binários não modificados como funções não-membros. Se um operador binário for definido como um membro de classe, conversões implícitas se aplicarão ao argumento da direita, mas não à mão esquerda. Confundirá seus usuários se compilar, mas não o faz.`a < bb < a`

Não saia do seu caminho para evitar definir sobre cargas de operadores. Por exemplo, prefira definir `,`, `e`, `em vez de`, `.` . Por outro lado, não defina sobre cargas de operadores só porque outras bibliotecas esperam por elas. Por exemplo, se o seu tipo não tem um pedido natural, mas você quer armazená-lo em um `,` , use um comparador personalizado em vez de sobre carregar `==<<Equals()CopyFrom()PrintTo()std::set<`

Não sobre carregue `,` (címula) ou unary `.` . Não sobre carregue, ou seja, não introduza literais definidos pelo usuário. Não use tais literais fornecidos por outros (incluindo a biblioteca padrão).`&&||,&operator""`

Os operadores de conversão de tipos são cobertos na seção em [conversões implícitas](#). O operador está coberto na seção sobre [construtores de cópias](#). A sobre carga para uso com fluxos é coberta na seção em [córegos](#). Veja também as regras sobre [sobre carga de função](#), que se aplicam também à sobre carga do operador.=<<

☞ Controle de acesso

Faça os dados membros das classes, a menos que sejam [constantes](#). Isso simplifica o raciocínio sobre invariantes, ao custo de alguma caldeira fácil na forma de acessórios (geralmente) se necessário.`private const`

Por razões técnicas, permitimos que os membros de dados de uma classe de fixação de teste definida em um arquivo `.cc` sejam ao usar [o Google Test](#). Se uma classe de fixação de teste for definida fora do arquivo `.cc`, ela será usada, por exemplo, em um arquivo `.h`, faça membros de dados `.protected``.private`

☞ Ordem de Declaração

Agrupar declarações semelhantes, colocando partes públicas mais cedo.

Uma definição de classe geralmente deve começar com uma seção, seguida por `,`, então `.` . Omitir seções que estariam vazias.`public:protected:private:`

Dentro de cada seção, prefira agrupar tipos semelhantes de declarações e prefira a seguinte ordem: tipos e codinomes (`,`, `,`, estruturas aninhadas e classes), constantes estáticas, funções de fábrica, construtores e operadores de atribuição, destruidor, todos os outros membros e funções, membros de dados.`typedefusingenumfriend`

Não coloque grandes definições de método em linha na definição de classe. Normalmente, apenas métodos triviais ou críticos de desempenho, e muito curtos, podem ser definidos inline. Consulte [As Funções Inline](#) para obter mais detalhes.

☞ Funções

↔ Entradas e saídas

A saída de uma função C++ é naturalmente fornecida através de um valor de retorno e, às vezes, através de parâmetros de saída (ou parâmetros de dentro/para fora).

Prefira usar valores de retorno em vez de parâmetros de saída: eles melhoram a legibilidade e, muitas vezes, fornecem o mesmo ou melhor desempenho.

Prefira retornar por valor ou, caso contrário, retorne por referência. Evite devolver um ponteiro a menos que possa ser nulo.

Os parâmetros são entradas para a função, saídas da função ou ambos. Os parâmetros de entrada não opcionais geralmente devem ser valores ou referências, enquanto parâmetros de saída e entrada/saída não opcionais geralmente devem ser referências (que não podem ser nulas). Geralmente, use para representar entradas opcionais por valor e use um ponteiro quando a forma não opcional teria usado uma referência. Use não-ponteiros para representar saídas opcionais e parâmetros opcionais de entrada/saída.`const std::optional<const const>`

Evite definir funções que requerem um parâmetro de referência para sobreviver à chamada, pois os parâmetros de referência se ligam a temporários. Em vez disso, encontre uma maneira de eliminar o requisito de vida (por exemplo, copiando o parâmetro), ou passá-lo por ponteiro e documentar os requisitos de vida e não nulo. `const const const`

Ao encomendar parâmetros de função, coloque todos os parâmetros somente de entrada antes de qualquer parâmetro de saída. Em particular, não adicione novos parâmetros ao final da função apenas porque eles são novos; coloque novos parâmetros somente de entrada antes dos parâmetros de saída. Esta não é uma regra difícil e rápida.

Parâmetros que são tanto de entrada quanto de saída enlameam as águas, e, como sempre, a consistência com funções relacionadas podem exigir que você dobre a regra. Funções variadas também podem exigir pedidos incomuns de parâmetros.

↔ Escrever funções curtas

Prefira funções pequenas e focadas.

Reconhecemos que funções longas às vezes são apropriadas, por isso não é colocado um limite rígido no comprimento das funções. Se uma função exceder cerca de 40 linhas, pense se ela pode ser quebrada sem prejudicar a estrutura do programa.

Mesmo que sua longa função funcione perfeitamente agora, alguém modificando-a em poucos meses pode adicionar um novo comportamento. Isso pode resultar em bugs difíceis de encontrar. Manter suas funções curtas e simples torna mais fácil para outras pessoas ler e modificar seu código. Funções pequenas também são mais fáceis de testar.

Você pode encontrar funções longas e complicadas ao trabalhar com algum código. Não se intimide modificando o código existente: se trabalhar com tal função se mostra difícil, você descobre que erros são difíceis de depurar, ou você quer usar um pedaço dele em vários contextos diferentes, considere dividir a função em peças menores e mais gerenciáveis.

↔ Sobrecarga de função

Use funções sobrecarregadas (incluindo construtores) somente se um leitor olhando para um site de chamadas pode ter uma boa ideia do que está acontecendo sem ter que primeiro descobrir exatamente qual sobrecarga está sendo chamada.

Definition:

Você pode escrever uma função que leva um e sobrecarregá-lo com outro que leva . No entanto, neste caso considere `std::string_view` em vez disso.`const std::string& const char*`

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

Pros:

A sobrecarga pode tornar o código mais intuitivo, permitindo que uma função com nome idêntico tome diferentes argumentos. Pode ser necessário para código templatizado, e pode ser conveniente para visitantes.

A sobrecarga com base na qualificação de const ou árbitro pode tornar o código de utilidade mais utilizável, mais eficiente ou ambos. (Consulte [TotW 148](#) para mais.)

Cons:

Se uma função estiver sobrecarregada apenas pelos tipos de argumento, o leitor pode ter que entender as complexas regras de correspondência do C++, a fim de dizer o que está acontecendo. Também muitas pessoas ficam confusas com a semântica da herança se uma classe derivada substitui apenas algumas das variantes de uma função.

Decision:

Você pode sobreregar uma função quando não há diferenças semânticas entre as variantes. Essas sobreargas podem variar em tipos, qualificações ou contagem de argumentos. No entanto, um leitor de tal chamada não deve saber qual membro do conjunto de sobrearga é escolhido, apenas que **algo** do conjunto está sendo chamado. Se você puder documentar todas as entradas no conjunto de sobrearga com um único comentário no cabeçalho, isso é um bom sinal de que é um conjunto de sobrearga bem projetado.

↳ Argumentos padrão

Argumentos padrão são permitidos em funções não virtuais quando o padrão é garantido para ter sempre o mesmo valor. Siga as mesmas restrições que para [sobrearga de função](#) e prefira funções sobrearregadas se a legibilidade obtida com argumentos padrão não superar as desvantagens abaixo.

Pros:

Muitas vezes você tem uma função que usa valores padrão, mas ocasionalmente você deseja substituir os padrões. Os parâmetros padrão permitem uma maneira fácil de fazer isso sem ter que definir muitas funções para as raras exceções. Em comparação com a sobrearga da função, os argumentos padrão têm uma sintaxe mais limpa, com menos caldeira e uma distinção mais clara entre argumentos 'necessários' e 'opcionais'.

Cons:

Argumentos padrão são outra maneira de alcançar a semântica das funções sobrearregadas, de modo que todas as [razões para não sobrearregar funções](#) se aplicam.

Os padrões para argumentos em uma chamada de função virtual são determinados pelo tipo estático do objeto alvo, e não há garantia de que todas as substituições de uma determinada função declarem os mesmos padrões.

Os parâmetros padrão são reavaliados em cada site de chamada, o que pode inchar o código gerado. Os leitores também podem esperar que o valor padrão seja fixado na declaração em vez de variar em cada chamada.

Os ponteiros de função são confusos na presença de argumentos padrão, uma vez que a assinatura da função muitas vezes não corresponde à assinatura de chamada. Adicionar sobreargas de função evita esses problemas.

Decision:

Os argumentos padrão são proibidos em funções virtuais, onde não funcionam corretamente, e nos casos em que o padrão especificado pode não avaliar o mesmo valor dependendo de quando foi avaliado. (Por exemplo, não escreva `.void f(int n = counter++);`)

Em alguns outros casos, os argumentos padrão podem melhorar a legibilidade de suas declarações de função o suficiente para superar as desvantagens acima, para que sejam permitidas. Na dúvida, use sobreargas.

↳ Sintaxe do tipo de retorno em movimento

Use os tipos de retorno de arrasto apenas quando usar a sintaxe comum (tipos de retorno principais) é impraticável ou muito menos legível.

Definition:

C++ permite duas formas diferentes de declarações de função. Na forma mais antiga, o tipo de retorno aparece antes do nome da função. Por exemplo:

```
int foo(int x);
```

O formulário mais recente usa a palavra-chave antes do nome da função e um tipo de retorno após a lista de argumentos. Por exemplo, a declaração acima poderia ser escrita equivalentemente:`auto`

```
auto foo(int x) -> int;
```

O tipo de retorno está no escopo da função. Isso não faz diferença para um caso simples como, mas importa para casos mais complicados, como tipos declarados no escopo de classe ou tipos escritos em termos dos parâmetros de função.`int`

Pros:

Os tipos de retorno de arrasto são a única maneira de especificar explicitamente o tipo de retorno de uma [expressão lambda](#). Em alguns casos, o compilador é capaz de deduzir o tipo de retorno de uma lambda, mas não em todos os casos. Mesmo quando o compilador pode deduzi-lo automaticamente, às vezes especificando-o explicitamente seria mais claro para os leitores.

Às vezes é mais fácil e mais legível especificar um tipo de retorno após a lista de parâmetros da função já ter aparecido. Isso é particularmente verdade quando o tipo de retorno depende dos parâmetros do modelo. Por exemplo:

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

contra

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

Cons:

A sintaxe do tipo de retorno é relativamente nova e não tem análogo em linguagens semelhantes a C++, como C e Java, então alguns leitores podem achar estranho.

As bases de código existentes têm um enorme número de declarações de função que não serão alteradas para usar a nova sintaxe, então as escolhas realistas estão usando apenas a sintaxe antiga ou usando uma mistura dos dois. Usar uma única versão é melhor para a uniformidade de estilo.

Decision:

Na maioria dos casos, continue usando o estilo mais antigo da declaração de função onde o tipo de retorno vai antes do nome da função. Use o novo formulário de retorno de trilha apenas nos casos em que for necessário (como lambdas) ou onde, colocando o tipo após a lista de parâmetros da função, ele permite que você escreva o tipo de forma muito mais legível. Este último caso deve ser raro; é principalmente um problema em código de modelo bastante complicado, que é [desencorajado na maioria dos casos](#).

↔ Magia específica do Google

Existem vários truques e utilitários que usamos para tornar o código C++ mais robusto, e várias maneiras que usamos C++ que podem diferir do que você vê em outros lugares.

↔ Propriedade e Ponteiros Inteligentes

Prefira ter proprietários fixos simples para objetos alocados dinamicamente. Prefira transferir a propriedade com ponteiros inteligentes.

Definition:

"Propriedade" é uma técnica de contabilidade para gerenciar a memória alocada dinamicamente (e outros recursos). O proprietário de um objeto alocado dinamicamente é um objeto ou função que é responsável por garantir que ele seja excluído quando não for mais necessário. A propriedade às vezes pode ser compartilhada, nesse caso o último proprietário é tipicamente responsável por excluí-la. Mesmo quando a propriedade não é compartilhada, ela pode ser transferida de um pedaço de código para outro.

Os ponteiros "inteligentes" são classes que agem como ponteiros, por exemplo, sobrecarregando os operadores. Alguns tipos de ponteiros inteligentes podem ser usados para automatizar a contabilidade de propriedade, para garantir que essas responsabilidades sejam cumpridas. [`std::unique_ptr`](#) é um tipo de ponteiro inteligente que expressa a propriedade exclusiva de um objeto alocado dinamicamente; o objeto é excluído quando o sai do escopo. Não pode ser copiado, mas pode ser *movido* para representar a transferência de propriedade. [`std::shared_ptr`](#) é um tipo de ponteiro inteligente que expressa a propriedade compartilhada de um objeto alocado dinamicamente. S pode ser copiado; a propriedade do objeto é compartilhada entre todas as cópias, e o objeto é excluído quando o último é destruído. `*->std::unique_ptr``std::shared_ptr`

Pros:

- É virtualmente impossível gerenciar a memória dinamicamente alocada sem algum tipo de lógica de propriedade.
- Transferir a propriedade de um objeto pode ser mais barato do que copiá-lo (se copiá-lo é mesmo possível).
- Transferir a propriedade pode ser mais simples do que 'emprestar' um ponteiro ou referência, porque reduz a necessidade de coordenar a vida útil do objeto entre os dois usuários.
- Os ponteiros inteligentes podem melhorar a legibilidade, tornando a lógica de propriedade explícita, auto-documentada e inequívoca.
- Os ponteiros inteligentes podem eliminar a contabilidade de propriedade manual, simplificando o código e excluindo grandes classes de erros.
- Para objetos const, a propriedade compartilhada pode ser uma alternativa simples e eficiente para copiar profundamente.

Cons:

- A propriedade deve ser representada e transferida através de ponteiros (sejam inteligentes ou simples). A semântica do ponteiro é mais complicada do que a semântica de valor, especialmente nas APIs: você tem que se preocupar não apenas com a propriedade, mas também com o aliasing, a vida vital e a mutabilidade, entre outras questões.
- Os custos de desempenho da semântica de valor são muitas vezes superestimados, de modo que os benefícios de desempenho da transferência de propriedade podem não justificar os custos de legibilidade e complexidade.
- As APIs que transferem a propriedade forçam seus clientes a um único modelo de gerenciamento de memória.
- O código usando ponteiros inteligentes é menos explícito sobre onde as liberações de recursos ocorrem.
- `std::unique_ptr` expressa transferência de propriedade usando semântica de movimento, que são relativamente novas e podem confundir alguns programadores.
- A propriedade compartilhada pode ser uma alternativa tentadora ao design cuidadoso da propriedade, ofuscando o design de um sistema.
- A propriedade compartilhada requer contabilidade explícita em tempo de execução, o que pode ser caro.
- Em alguns casos (por exemplo, referências cíclicas), objetos com propriedade compartilhada podem nunca ser excluídos.

- Ponteiros inteligentes não são substitutos perfeitos para ponteiros simples.

Decision:

Se a alocação dinâmica for necessária, prefira manter a propriedade com o código que o alocou. Se outro código precisar de acesso ao objeto, considere passar uma cópia ou passar um ponteiro ou referência sem transferir a propriedade. Prefira usar para tornar a transferência de propriedade explícita. Por exemplo:`std::unique_ptr`

```
std::unique_ptr<Foo> FooFactory();
void FooConsumer(std::unique_ptr<Foo> ptr);
```

Não desenhe seu código para usar a propriedade compartilhada sem uma razão muito boa. Uma dessas razões é evitar operações de cópia caras, mas você só deve fazer isso se os benefícios de desempenho forem significativos, e o objeto subjacente for imutável (ou seja,). Se você usar propriedade compartilhada, prefira usar `.std::shared_ptr<const Foo> std::shared_ptr`

Nunca use . Em vez disso, use `.std::auto_ptr std::unique_ptr`

☞ cpplint

Use para detectar erros de estilo.`cpplint.py`

`cpplint.py` é uma ferramenta que lê um arquivo de origem e identifica muitos erros de estilo. Não é perfeito, e tem falsos positivos e falsos negativos, mas ainda é uma ferramenta valiosa.

Alguns projetos têm instruções sobre como executar a partir de suas ferramentas de projeto. Se o projeto para o que você está contribuindo não, você pode baixar [cpplint.py](#) separadamente.`cpplint.py`

☞ Outros recursos C++

☞ Referências de Rvalue

Use referências de rvalue apenas em certos casos especiais listados abaixo.

Definition:

Referências de rvalue são um tipo de referência que só pode se ligar a objetos temporários. A sintaxe é semelhante à sintaxe de referência tradicional. Por exemplo, declara uma função cujo argumento é uma referência rvalue a um `std::string`.

Quando o token '`&&`' é aplicado a um argumento de modelo não qualificado em um parâmetro de função, regras especiais de dedução de argumento de modelo se aplicam. Essa referência é chamada de referência de encaminhamento.

Pros:

- Definir um construtor de movimento (um construtor que faz uma referência de rvalue ao tipo de classe) torna possível mover um valor em vez de copiá-lo. Se for um , por exemplo, provavelmente resultará apenas em alguma simples manipulação de ponteiro em vez de copiar uma grande quantidade de dados. Em muitos casos, isso pode resultar em uma grande melhoria de desempenho.
- `v1 std::vector<std::string> auto v2(std::move(v1))`
- As referências de Rvalue possibilitam a implementação de tipos móveis, mas não copiáveis, que podem ser úteis para tipos que não têm definição sensata de cópia, mas onde você ainda pode querer passá-los como argumentos de função, colocá-los em recipientes, etc.
- `std::move` é necessário fazer uso efetivo de alguns tipos de biblioteca padrão, tais como `.std::unique_ptr`
- [Encaminhamento de referências](#) que usam o token de referência rvalue, possibilita a gravação de um invólucro de função genérico que encaminha seus argumentos para outra função e funciona se seus argumentos são ou não objetos temporários e/ou const. Isso é chamado de "encaminhamento perfeito".

Cons:

- As referências de rvalue ainda não são amplamente compreendidas. Regras como o colapso de referência e a regra especial de dedução para encaminhamento de referências são um pouco obscuras.
- As referências de rvalue são muitas vezes mal utilizadas. O uso de referências de rvalue é contra-intuitivo em assinaturas onde se espera que o argumento tenha um estado especificado válido após a chamada de função ou onde nenhuma operação de movimento é realizada.

Decision:

Não utilize referências de rvalue (ou aplique o qualificatório aos métodos), exceto da seguinte forma:`&&`

- Você pode usá-los para definir construtores de movimento e mover operadores de atribuição (conforme descrito em [Tipos Copiáveis e Móveis](#)).
- Você pode usá-los para definir métodos qualificados que logicamente "consomem", deixando-os em um estado inutilizável ou vazio. Note que isso se aplica apenas às qualificatórias de método (que vêm após o parêntese de

- fechamento da assinatura da função); se você quiser "consumir" um parâmetro de função comum, prefira passá-lo por valor.`&&*this`
- Você pode usar referências de encaminhamento em conjunto com `, para suportar o encaminhamento perfeito.`
`std::forward`
 - Você pode usá-los para definir pares de sobrecargas, como uma tomada e outra tomada . Geralmente a solução preferida é apenas passar por valor, mas um par de funções sobrecarregadas às vezes produz melhor desempenho e às vezes é necessário em código genérico que precisa suportar uma grande variedade de tipos. Como sempre: se você está escrevendo códigos mais complicados por causa do desempenho, certifique-se de ter evidências de que isso realmente ajuda.`Foo&const Foo&`

↔ Amigos

Permitimos o uso de aulas e funções, dentro da razão.`friend`

Os amigos geralmente devem ser definidos no mesmo arquivo para que o leitor não precise procurar em outro arquivo para encontrar usos dos membros privados de uma classe. Um uso comum é ter uma classe amiga para que ela possa construir o estado interno de corretamente, sem expor este estado ao mundo. Em alguns casos, pode ser útil fazer de uma classe unittest um amigo da classe que testa.`friend FooBuilder Foo Foo`

Amigos estendem, mas não quebram, o limite de encapsulamento de uma classe. Em alguns casos, isso é melhor do que tornar um membro público quando você quer dar apenas uma outra classe acesso a ele. No entanto, a maioria das aulas deve interagir com outras classes apenas através de seus membros públicos.

↔ Exceções

Não usamos exceções C++.

Pros:

- As exceções permitem que níveis mais altos de um aplicativo decidam como lidar com falhas "não podem acontecer" em funções profundamente aninhadas, sem a ocultação e a reserva propensa a erros de códigos de erro.
- Exceções são usadas pela maioria das outras línguas modernas. Usá-los em C++ tornaria mais consistente com Python, Java e o C++ que outros estão familiarizados.
- Algumas bibliotecas C++ de terceiros usam exceções, e desligá-las internamente torna mais difícil integrar-se com essas bibliotecas.
- Exceções são a única maneira de um construtor falhar. Podemos simular isso com uma função de fábrica ou um método, mas estes requerem alocação de pilhas ou um novo estado "inválido", respectivamente.`Init()`
- Exceções são realmente úteis em estruturas de teste.

Cons:

- Quando você adiciona uma declaração a uma função existente, você deve examinar todos os seus chamadores transitivos. Ou eles devem fazer pelo menos a garantia básica de segurança de exceção, ou eles nunca devem pegar a exceção e estar felizes com o término do programa como resultado. Por exemplo, se chama chamadas , e lança uma exceção que pega, tem que ter cuidado ou pode não limpar corretamente.`throwf() g() h() hfg`
- De forma mais geral, as exceções dificultam o fluxo de controle de programas olhando para o código: as funções podem retornar em lugares que você não espera. Isso causa dificuldades de manutenção e depuração. Você pode minimizar esse custo através de algumas regras sobre como e onde exceções podem ser usadas, mas ao custo de mais que um desenvolvedor precisa saber e entender.
- A segurança de exceção requer raii e diferentes práticas de codificação. Muitas máquinas de suporte são necessárias para tornar a escrita um código seguro para exceções correta facilmente. Além disso, para evitar exigir que os leitores entendam todo o gráfico de chamadas, o código seguro para exceções deve isolar a lógica que escreve para o estado persistente em uma fase de "compromisso". Isso terá tanto benefícios quanto custos (talvez onde você é forçado a obfuscate o código para isolar o compromisso). Permitir exceções nos faria pagar sempre esses custos mesmo quando eles não valem a pena.
- Ligar exceções adiciona dados a cada binário produzido, aumentando o tempo de compilação (provavelmente ligeiramente) e possivelmente aumentando a pressão do espaço de endereço.
- A disponibilidade de exceções pode encorajar os desenvolvedores a jogá-las quando não forem apropriados ou se recuperar deles quando não é seguro fazê-lo. Por exemplo, a entrada inválida do usuário não deve causar exceções. Precisaríamos fazer o guia de estilo ainda mais longo para documentar essas restrições!

Decision:

Na sua cara, os benefícios do uso de exceções superam os custos, especialmente em novos projetos. No entanto, para o código existente, a introdução de exceções tem implicações em todos os códigos dependentes. Se exceções podem ser propagadas além de um novo projeto, também se torna problemático integrar o novo projeto ao código livre de exceções existente. Como a maioria dos códigos C++ existentes no Google não está preparada para lidar com exceções, é comparativamente difícil adotar um novo código que gera exceções.

Dado que o código existente do Google não é tolerante a exceções, os custos de uso de exceções são um pouco maiores do que os custos em um novo projeto. O processo de conversão seria lento e propenso a erros. Não acreditamos que as alternativas disponíveis para exceções, como códigos de erro e afirmações, introduzem um fardo significativo.

Nosso conselho contra o uso de exceções não se baseia em motivos filosóficos ou morais, mas práticos. Como gostaríamos de usar nossos projetos de código aberto no Google e é difícil fazê-lo se esses projetos usarem exceções, precisamos aconselhar contra exceções em projetos de código aberto do Google também. As coisas provavelmente seriam diferentes se tivéssemos que fazer tudo de novo do zero.

Essa proibição também se aplica a características relacionadas ao manuseio de exceções, como e .std::exception_ptr std::nested_exception

Há uma [exceção](#) a esta regra (sem trocadilhos) para o código do Windows.

☞ noexcept

Especifique quando é útil e correto.noexcept

Definition:

O especificador é usado para especificar se uma função irá lançar exceções ou não. Se uma exceção escapar de uma função marcada, o programa falha através de .noexcept noexcept std::terminate

O operador realiza uma verificação de tempo de compilação que retorna verdadeira se uma expressão for declarada para não lançar exceções.noexcept

Pros:

- Especificar construtores de movimentos como melhora o desempenho em alguns casos, por exemplo, move-se em vez de copiar os objetos se o construtor de movimento de T for .noexcept std::vector<T>::resize() noexcept
- Especificar em uma função pode desencadear otimizações de compiladores em ambientes onde exceções são ativadas, por exemplo, o compilador não precisa gerar código extra para desenrolar pilhas, se ele sabe que nenhuma exceção pode ser lançada devido a um especificador.noexcept noexcept

Cons:

- Em projetos que seguem este guia que têm exceções desativadas é difícil garantir que os especificadores estejam corretos e difícil definir o que a correção significa mesmo.noexcept
- É difícil, se não impossível, desfazer porque elimina uma garantia de que os chamadores podem estar confiando, de maneiras difíceis de detectar.noexcept

Decision:

Você pode usar quando é útil para o desempenho se refletir com precisão a semântica pretendida de sua função, ou seja, que se uma exceção é de alguma forma lançada de dentro do corpo da função, então ela representa um erro fatal. Você pode assumir que em movimento os construtores tem um benefício significativo de desempenho. Se você acha que há um benefício significativo de desempenho de especificar em alguma outra função, por favor, discuta isso com os leads do seu projeto.noexcept noexcept noexcept

Prefira incondicionalmente se as exceções forem completamente desativadas (ou seja, a maioria dos ambientes Do Google C++). Caso contrário, use especificadores condicionais com condições simples, de maneiras que avaliam falsas apenas nos poucos casos em que a função poderia potencialmente jogar. Os testes podem incluir características de tipo verificar se a operação envolvida pode ser jogada (por exemplo, para objetos de construção de movimento) ou se a alocação pode ser jogada (por exemplo, para alocação padrão padrão padrão). Note que, em muitos casos, a única causa possível para uma exceção é a falha de alocação (acreditamos que os construtores de movimento não devem jogar exceto devido à falha de alocação), e há muitas aplicações onde é apropriado tratar a exaustão da memória como um erro fatal, em vez de uma condição excepcional da qual seu programa deve tentar se recuperar. Mesmo para outras falhas potenciais, você deve priorizar a simplicidade da interface em vez de suportar todos os possíveis cenários de lançamento de exceções: em vez de escrever uma cláusula complicada que depende se uma função hash pode lançar, por exemplo, simplesmente documentar que seu componente não suporta funções de hash jogando e torná-lo incondicionalmente .noexcept noexcept std::is_nothrow_move_constructible abs!::default_allocator_is_nothrow noexcept noexcept

☞ Informações do tipo de tempo de execução (RTTI)

Evite usar informações do tipo run-time (RTTI).

Definition:

O RTTI permite que um programador consulte a classe C++ de um objeto no tempo de execução. Isso é feito por uso de ou .typeid dynamic_cast

Pros:

As alternativas padrão ao RTTI (descrito abaixo) requerem modificação ou redesenho da hierarquia de classe em questão. Às vezes, tais modificações são inviáveis ou indesejáveis, particularmente em códigos amplamente utilizados ou maduros.

RTTI pode ser útil em alguns testes unitários. Por exemplo, é útil em testes de classes de fábrica onde o teste tem que verificar se um objeto recém-criado tem o tipo dinâmico esperado. Também é útil na gestão da relação entre objetos e seus simulados.

RTTI é útil quando se considera múltiplos objetos abstratos. Considerar

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

```
 } ...
```

Cons:

Consultar o tipo de objeto em tempo de execução significa frequentemente um problema de design. Precisar saber o tipo de objeto no tempo de execução é muitas vezes uma indicação de que o design da hierarquia de classe é falho.

O uso indisciplinado do RTTI torna o código difícil de manter. Pode levar a árvores de decisão baseadas em tipo ou trocar declarações espalhadas por todo o código, todas as quais devem ser examinadas ao fazer novas alterações.

Decision:

RTTI tem usos legítimos, mas é propenso a abusos, então você deve ter cuidado ao usá-lo. Você pode usá-lo livremente em testes unitários, mas evite-o quando possível em outro código. Em particular, pense duas vezes antes de usar RTTI em novo código. Se você se encontrar precisando escrever um código que se comporte de forma diferente com base na classe de um objeto, considere uma das seguintes alternativas para consultar o tipo:

- Os métodos virtuais são a maneira preferida de executar diferentes caminhos de código, dependendo de um tipo específico de subclasse. Isso coloca o trabalho dentro do próprio objeto.
- Se o trabalho pertencer fora do objeto e, em vez disso, em algum código de processamento, considere uma solução de despacho duplo, como o padrão de design do Visitante. Isso permite que uma instalação fora do próprio objeto determine o tipo de classe usando o sistema de tipo embutido.

Quando a lógica de um programa garante que uma determinada instância de uma classe base é de fato uma instância de uma determinada classe derivada, então um pode ser usado livremente no objeto. Normalmente, pode-se usar como alternativa em tais situações.`dynamic_cast`

As árvores de decisão baseadas no tipo são uma forte indicação de que seu código está no caminho errado.

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

Código como este geralmente quebra quando subclasses adicionais são adicionadas à hierarquia de classe. Além disso, quando as propriedades de uma subclasse mudam, é difícil encontrar e modificar todos os segmentos de código afetados.

Não implemente manualmente uma solução alternativa semelhante a RTTI. Os argumentos contra o RTTI aplicam-se tanto às soluções alternativas como hierarquias de classe com tags de tipo. Além disso, soluções alternativas disfarçam sua verdadeira intenção.

☞ Fundição

Use moldes no estilo C++, ou inicialização de suporte para conversão de tipos aritméticos como . Não use formatos de elenco como a menos que o elenco seja. Você só pode usar formatos de elenco como 'T(x)' quando 'T' é um tipo de classe.`static_cast<float>(double_value)``int64_t y = int64_t{1} << 42(int)x``void`

Definition:

C++ introduziu um sistema de elenco diferente de C que distingue os tipos de operações de elenco.

Pros:

O problema com os moldes C é a ambiguidade da operação; às vezes você está fazendo uma conversão (por exemplo), e às vezes você está fazendo um gesso (por exemplo,). Inicialização da cinta e moldes C++ muitas vezes podem ajudar a evitar essa ambiguidade. Além disso, os moldes C++ são mais visíveis ao procurá-los.`(int)3.5(int)"hello"`

Cons:

A sintaxe de elenco estilo C++ é verbosa e complicada.

Decision:

Em geral, não use moldes estilo C. Em vez disso, use esses moldes estilo C++ quando for necessária a conversão de tipo explícita.

- Use a inicialização da cinta para converter tipos aritméticos (por exemplo,). Esta é a abordagem mais segura porque o código não compilará se a conversão pode resultar em perda de informações. A sintaxe também é concisa.`int64_t{x}`
- Use para lançar com segurança uma hierarquia de tipo, por exemplo, lançando a a ou lançando a a . C++ geralmente faz isso automaticamente, mas algumas situações precisam de um up-cast explícito, como o uso do operador.`abs1::implicit_cast<Foo*>Superclass0::Foo*const Foo*?;`
- Use como equivalente a um elenco estilo C que faça a conversão de valor, quando você precisa explicitamente levantar um ponteiro de uma classe para sua superclasse, ou quando você precisa lançar explicitamente um ponteiro de uma superclasse para uma subclasse. Neste último caso, você deve ter certeza de que seu objeto é realmente uma instância da subclasse.`static_cast`
- Use para remover o qualificador (ver `const`).`const_cast`
- Use para fazer conversões insecuras de tipos de ponteiros de e para inteiros e outros tipos de ponteiros, incluindo . Use isso apenas se você souber o que está fazendo e entender os problemas de aliasing. Além disso,

- considere a alternativa.`reinterpret_cast<void*>::bit_cast`
- Use para interpretar os bits brutos de um valor usando um tipo diferente do mesmo tamanho (um trocadilho tipo), como interpretar os bits de um como `.absl::bit_cast<double>int64_t`

Consulte a [seção RTTI](#) para orientação sobre o uso de `.dynamic_cast`

☞ Fluxos

Use fluxos quando apropriado e atenha-se a usos "simples". Sobrecarga para streaming apenas para tipos que representam valores e escreva apenas o valor visível do usuário, não quaisquer detalhes de implementação.<<

Definition:

Os fluxos são a abstração padrão de I/O em C++, conforme exemplificado pelo cabeçalho padrão `<iostream>`. Eles são amplamente utilizados no código do Google, principalmente para depuração de registros e diagnósticos de teste.<<>>`printf``std::string``printf`

Pros:

Os operadores e stream fornecem uma API para I/O formatado que é facilmente aprendida, portátil, reutilizável e extensível., por outro lado, nem sequer suporta , para não dizer nada de tipos definidos pelo usuário, e é muito difícil de usar portavelmente. também o obriga a escolher entre as inúmeras versões ligeiramente diferentes dessa função, e navegar pelas dezenas de especificadores de conversão.<<>>`printf``std::cout``std::cerr``std::clog`

Cons:

- A formatação do fluxo pode ser configurada mutando o estado do fluxo. Tais mutações são persistentes, de modo que o comportamento do seu código pode ser afetado por toda a história anterior do fluxo, a menos que você saia do seu caminho para restaurá-lo a um estado conhecido cada vez que outro código pode ter tocado nele. O código do usuário não só pode modificar o estado interno, como pode adicionar novas variáveis e comportamentos de estado através de um sistema de registro.
- É difícil controlar precisamente a saída do fluxo, devido aos problemas acima, a forma como o código e os dados são misturados no código de streaming e o uso da sobrecarga do operador (que pode selecionar uma sobrecarga diferente do que você espera).
- A prática de construir a produção através de cadeias de operadores interfere na internacionalização, pois assa a ordem das palavras no código, e o suporte dos fluxos para localização é [falso](#).<<
- A API de fluxos é útil e complexa, por isso os programadores devem desenvolver experiência com ela para usá-la de forma eficaz.
- Resolver as muitas sobrecargas é extremamente caro para o compilador. Quando usado de forma generalizada em uma grande base de código, pode consumir até 20% do tempo de análise e análise semântica.<<

Decision:

Use fluxos somente quando eles são a melhor ferramenta para o trabalho. Este é tipicamente o caso quando o I/O é ad-hoc, local, ser lêvel e direcionado a outros desenvolvedores em vez de usuários finais. Seja consistente com o código ao seu redor, e com a base de código como um todo; se houver uma ferramenta estabelecida para o seu problema, use essa ferramenta em vez disso. Em particular, bibliotecas de registro geralmente são uma escolha melhor do que ou para saída de diagnóstico, e as bibliotecas em ou o equivalente são geralmente uma escolha melhor do que `.std::cerr``std::clog``absl::strings``std::stringstream`

Evite o uso de fluxos para I/O que enfrente usuários externos ou manuseie dados não confiáveis. Em vez disso, encontre e use as bibliotecas de templating apropriadas para lidar com questões como internacionalização, localização e endurecimento da segurança.

Se você usar fluxos, evite as partes imponentes da API de fluxos (além do estado de erro), tais como , e . Use funções de formatação explícitas (como) em vez de manipuladores de fluxo ou sinalizadores de formatação para controlar detalhes de formatação, como base de números, precisão ou preenchimento.`imbue()``xalloc()``register_callback()``absl::StreamFormat()`

Sobrecarregue como um operador de streaming para o seu tipo apenas se o seu tipo representa um valor, e escreva uma representação de sequência de sequências legível por humanos desse valor. Evite expor detalhes de implementação na saída de; se você precisar imprimir internos de objetos para depuração, use funções nomeadas em vez disso (um método nomeado é a convenção mais comum).<<<<<<`DebugString()`

☞ Pré-cremento e Pós-cremento

Use a forma de prefixo () dos operadores de incremento e decréscrito, a menos que você precise de semântica pós-fixação.`++i`

Definition:

Quando uma variável é incrementada (ou) ou decrépida (ou) e o valor da expressão não é utilizado, deve-se decidir se o pré-cre (decréspio) ou o pós-cre (decrésia).`++i``++i``--i``--i`

Pros:

Uma expressão de incremento/decréscrito pós-fixado avalia o valor *como era antes de ser modificado*. Isso pode resultar em código mais compacto, mas mais difícil de ler. O formulário de prefixo é geralmente mais legível, nunca é

menos eficiente, e pode ser mais eficiente porque não precisa fazer uma cópia do valor como era antes da operação.

Cons:

A tradição desenvolveu-se, em C, o uso pós-incremento, mesmo quando o valor de expressão não é utilizado, especialmente em loops.`for`

Decision:

Use incremento/decréscreção do prefixo, a menos que o código precise explicitamente do resultado da expressão de incremento/decrésrito pós-fixado.

☞ Uso de const

Em APIs, use sempre que fizer sentido. É uma escolha melhor para alguns usos de `const`, `constexpr`

Definition:

As variáveis e parâmetros declarados podem ser precedidos pela palavra-chave para indicar que as variáveis não foram alteradas (por exemplo, `const`). As funções de classe podem ter o qualificatório para indicar que a função não altera o estado das variáveis dos membros da classe (por exemplo, `const`).

```
.constconst int fooconstclass Foo { int Bar(char c) const; };
```

Pros:

É mais fácil para as pessoas entenderem como as variáveis estão sendo usadas. Permite que o compilador faça melhor a verificação do tipo e, conceitivamente, gerar um código melhor. Ajuda as pessoas a se convencerem da correção do programa porque sabem que as funções que chamam são limitadas em como podem modificar suas variáveis. Ajuda as pessoas a saber quais funções são seguras de usar sem fechaduras em programas multi-threaded.

Cons:

`const` é viral: se você passar uma variável para uma função, essa função deve ter em seu protótipo (ou a variável precisará de um `const`). Isso pode ser um problema particular ao chamar as funções da biblioteca `constconstconst_cast`

Decision:

Recomendamos fortemente o uso em APIs (ou seja, em parâmetros de função, métodos e variáveis não locais) onde quer que seja significativo e preciso. Isso fornece uma documentação consistente, principalmente verificada pelo compilador, de quais objetos uma operação pode sofrer mutação. Ter uma maneira consistente e confiável de distinguir leituras de gravações é fundamental para escrever código seguro para tópicos, e é útil em muitos outros contextos também. Em particular: `const`

- Se uma função garantir que não modificará um argumento passado por referência ou por ponteiro, o parâmetro de função correspondente deve ser uma referência ao `const` () ou ponteiro-a-`const` (respectivamente). `const T&const T*`
- Para um parâmetro de função passado por valor, não tem efeito sobre o chamador, portanto não é recomendado em declarações de função. Veja [TotW #109](#). `const`
- Declare os métodos a menos que alterem o estado lógico do objeto (ou permitam que o usuário modifique esse estado, por exemplo, retornando uma referência não-`const`, mas isso é raro), ou eles não podem ser invocados simultaneamente. `const`

O uso em variáveis locais não é encorajado nem desencorajado. `const`

Todas as operações de uma classe devem ser seguras para invocar simultaneamente entre si. Se isso não for viável, a classe deve ser claramente documentada como "insegura". `const`

Onde colocar o const

Algumas pessoas preferem o formulário para. Eles argumentam que isso é mais legível porque é mais consistente: mantém a regra que sempre segue o objeto que está descrevendo. No entanto, este argumento de consistência não se aplica em bases de código com poucas expressões de ponteiro profundamente aninhadas, uma vez que a maioria das expressões tem apenas uma , e se aplica ao valor subjacente. Nesses casos, não há consistência para manter. Colocar o primeiro é indiscutivelmente mais legível, uma vez que segue o inglês em colocar o "adjetivo" () antes do "substantivo" (). `int const *fooconst int* fooconstconstconstconstconstint`

Dito isso, enquanto encorajamos colocar em primeiro lugar, não exigimos isso. Mas seja consistente com o código ao seu redor! `const`

☞ Uso de constexpr

Use para definir constantes verdadeiras ou para garantir a inicialização constante. `constexpr`

Definition:

Algumas variáveis podem ser declaradas para indicar que as variáveis são constantes verdadeiras, ou seja, fixadas na hora da compilação/link. Algumas funções e construtores podem ser declarados, o que permite que eles sejam usados

na definição de uma variável.`constexprconstexprconstexpr`

Pros:

O uso permite a definição de constantes com expressões de ponto flutuante em vez de apenas literais; definição de constantes de tipos definidos pelo usuário; e definição de constantes com chamadas de função.`constexpr`

Cons:

Marcar prematuramente algo como `constexpr` pode causar problemas migratórios se mais tarde tiver que ser rebaixado. As restrições atuais sobre o que é permitido em funções `constexpr` e construtores podem convidar soluções obscuras nessas definições.

Decision:

`constexpr` definições permitem uma especificação mais robusta das partes constantes de uma interface. Use para especificar as constantes verdadeiras e as funções que suportam suas definições. Evite a complexificação de definições de função para habilitar seu uso com . Não use para forçar a inlineing.`constexprconstexprconstexpr`

↔ Tipos inteiros

Dos tipos inteiros C++ embutidos, o único usado é . Se um programa precisar de uma variável de um tamanho diferente, use um tipo inteiro de largura precisa de, como . Se sua variável representar um valor que pode ser maior ou igual a 2^{31} (2GiB), use um tipo de 64 bits como . Tenha em mente que, mesmo que seu valor nunca seja muito grande para um, ele pode ser usado em cálculos intermediários que podem exigir um tipo maior. Na dúvida, escolha um tipo maior.`int<cstdint>int16_tint64_tint`

Definition:

C++ não especifica com precisão os tamanhos de tipos inteiros como . Normalmente as pessoas assumem que é 16 bits, é 32 bits, é 32 bits e é 64 bits.`intshortintlonglong long`

Pros:

Uniformidade da declaração.

Cons:

Os tamanhos dos tipos inteiros em C++ podem variar de acordo com o compilador e a arquitetura.

Decision:

Usamos muitas vezes, para inteiros que sabemos que não serão muito grandes, por exemplo, contadores de loop. Use o velho para essas coisas. Você deve assumir que um é pelo menos 32 bits, mas não assuma que ele tem mais de 32 bits. Se você precisar de um tipo inteiro de 64 bits, use ou .`intintintint64_tuint64_t`

Para inteiros que sabemos que pode ser "grande", use .`int64_t`

Você não deve usar os tipos inteiros não assinados, como , a menos que haja uma razão válida, como representar um padrão de bit em vez de um número, ou você precisa de módulo de transbordamento definido 2^N . Em particular, não use tipos não assinados para dizer que um número nunca será negativo. Em vez disso, use afirmações para isso.`uint32_t`

Se o seu código for um recipiente que retorna um tamanho, certifique-se de usar um tipo que irá acomodar qualquer uso possível do seu recipiente. Na dúvida, use um tipo maior em vez de um tipo menor.

Use cuidado ao converter tipos inteiros. Conversões e promoções `integer` podem causar comportamentos indefinidos, levando a bugs de segurança e outros problemas.

Em Inteiros Não Assinados

Inteiros não assinados são bons para representar bitfields e aritmética modular. Por causa de acidentes históricos, o padrão C++ também usa inteiros não assinados para representar o tamanho dos contêineres - muitos membros do órgão de padrões acreditam que isso seja um erro, mas é efetivamente impossível corrigir neste momento. O fato de que a aritmética não assinada não modela o comportamento de um simples inteiro, mas é definido pelo padrão para modelar aritmética modular (envolvendo-se em transbordamento/subfluxo), significa que uma classe significativa de insetos não pode ser diagnosticada pelo compilador. Em outros casos, o comportamento definido impede a otimização.

Dito isto, a mistura de autografia de tipos inteiros é responsável por uma classe igualmente grande de problemas. O melhor conselho que podemos fornecer: tente usar iteradores e recipientes em vez de ponteiros e tamanhos, tente não misturar a assinatura e tente evitar tipos não assinados (exceto por representar bitfields ou aritmética modular). Não utilize um tipo não assinado apenas para afirmar que uma variável não é negativa.

↔ Portabilidade de 64 bits

O código deve ser 64 bits e 32 bits amigável. Tenha em mente problemas de impressão, comparações e alinhamento de estruturas.

- Os especificadores de conversão portáteis corretos para alguns `typedefs` integrais dependem de expansões macro que achamos desagradáveis de usar e impraticáveis para exigir (as macros de). A menos que não haja alternativa razoável para o seu caso específico, tente evitar ou mesmo atualizar APIs que dependem da família. Em vez disso, use uma biblioteca de suporte a formatação numérica, como `StrCat` ou `Substituir` para conversões simples rápidas, ou `std::ostream.printf()PRI<cinttypes>printf`

Infelizmente, as macros são a única maneira portátil de especificar uma conversão para os dígitos de largura de bit padrão padrão (por exemplo, , , , , etc). Sempre que possível, evite passar argumentos de tipos especificados por digitações de bitwidth para APIs baseadas. Observe que é aceitável usar `typedefs` para os quais a `printf` possui modificadores de comprimento dedicados, tais como () e ()`PRIint64_t``uint64_t``int32_t``tprintfsize_tzptrdiff_ttmaxint_tj`

- Lembre-se que != . Use se quiser um inteiro do tamanho de um ponteiro.`sizeof(void *)``sizeof(int)``intptr_t`
- Você pode precisar ter cuidado com os alinhamentos da estrutura, especialmente para estruturas armazenadas em disco. Qualquer classe/estrutura com um /membro será por padrão acabar sendo 8 byte alinhado em um sistema de 64 bits. Se você tiver essas estruturas sendo compartilhadas em disco entre código de 32 bits e 64 bits, você precisará garantir que elas estejam embaladas da mesma forma em ambas as arquiteturas. A maioria dos compiladores oferece uma maneira de alterar o alinhamento da estrutura. Para gcc, você pode usar . MSVC oferece e `.int64_t``uint64_t``_attribute__((packed))#pragma pack()``declspec(align())`
- Use a [inicialização do aparelho](#) conforme necessário para criar constantes de 64 bits. Por exemplo:

```
int64_t my_value{0x123456789};
uint64_t my_mask{ULL << 48};
```

⇒ Macros do pré-processador

Evite definir macros, especialmente em cabeçalhos; preferem funções inline, enums e variáveis. Nomeie macros com um prefixo específico do projeto. Não use macros para definir peças de uma API C++.

Macros significam que o código que você vê não é o mesmo que o código que o compilador vê. Isso pode introduzir comportamentos inesperados, especialmente porque as macros têm escopo global.

Os problemas introduzidos pelas macros são especialmente graves quando são usados para definir peças de uma API C++, e ainda mais para APIs públicas. Cada mensagem de erro do compilador quando os desenvolvedores usam incorretamente essa interface agora deve explicar como as macros formaram a interface. As ferramentas de refator e análise têm um tempo dramaticamente mais difícil de atualizar a interface. Como consequência, não permitimos especificamente o uso de macros dessa forma. Por exemplo, evite padrões como:

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

public:
    EXPAND_PUBLIC_WOMBAT_API(Foo)

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

Felizmente, as macros não são tão necessárias em C++ quanto em C. Em vez de usar uma macro para inline código crítico de desempenho, use uma função inline. Em vez de usar uma macro para armazenar uma constante, use uma variável. Em vez de usar uma macro para "abreviar" um nome de variável longa, use uma referência. Em vez de usar uma macro para compilar condicionalmente o código ... bem, não faça isso de jeito nenhum (exceto, é claro, para os guardas evitar a dupla inclusão de arquivos de cabeçalho). Torna os testes muito mais difíceis.

Macros podem fazer coisas que essas outras técnicas não podem, e você as vê na base de código, especialmente nas bibliotecas de nível inferior. E algumas de suas características especiais (como stringifying, concatenation, e assim por diante) não estão disponíveis através do idioma propriamente dito. Mas antes de usar uma macro, considere cuidadosamente se há uma maneira não macro de alcançar o mesmo resultado. Se você precisar usar uma macro para definir uma interface, entre em contato com seu projeto, o projeto leva a solicitar uma renúncia desta regra.

O seguinte padrão de uso evitará muitos problemas com macros; se você usar macros, siga-a sempre que possível:

- Não defina macros em um arquivo..h
- `#define` macros logo antes de usá-los, e eles logo depois.`#undef`
- Não apenas uma macro existente antes de substituí-la pela sua própria; em vez disso, escolha um nome que provavelmente seja único.`#undef`
- Tente não usar macros que se expandam para construções C++ desequilibrada, ou pelo menos documentar bem esse comportamento.
- Prefira não usar para gerar nomes de função/classe/variável.##

Exportar macros de cabeçalhos (ou seja, defini-los em um cabeçalho sem escorá-los antes do fim do cabeçalho) é extremamente desanimado. Se você exportar uma macro de um cabeçalho, ela deve ter um nome globalmente único. Para isso, ele deve ser nomeado com um prefixo que consiste no nome do seu projeto (mas maiústo). `#undef`

⇒ 0 e nullptr/NUL

Use para ponteiros, e para chars (e não para o literal)`.nullptr'\0'0`

Para ponteiros (valores de endereço), use , pois isso fornece tipo de segurança.`nullptr`

Para projetos C++03, prefira . Embora os valores sejam equivalentes, parece mais um ponteiro para o leitor, e alguns compiladores C++ fornecem definições especiais das quais permitem que eles forneçam avisos úteis. Nunca use para valores numéricos (inteiros ou flutuantes).`NULL0NULLNULLNULL`

Use para o caractere nulo. O uso do tipo correto torna o código mais legível. '\0'

☞ tamanhos de

Prefira.`sizeof(varname)``sizeof(type)`

Use quando você pegar o tamanho de uma determinada variável. atualizará adequadamente se alguém mudar o tipo de variável agora ou mais tarde. Você pode usar para código não relacionado a qualquer variável específica, como código que gerencia um formato de dados externo ou interno onde uma variável de um tipo C++ apropriado não é conveniente.`sizeof(varname)``sizeof(varname)``sizeof(type)`

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

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

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

☞ Dedução de tipo (incluindo auto)

Use a dedução do tipo somente se ele deixar o código mais claro para os leitores que não estão familiarizados com o projeto, ou se torna o código mais seguro. Não o use apenas para evitar o inconveniente de escrever um tipo explícito.

Definition:

Existem vários contextos em que c++ permite (ou mesmo requer) que os tipos sejam deduzidos pelo compilador, em vez de explicitados explicitamente no código:

Dedução do argumento do modelo de função

Um modelo de função pode ser invocado sem argumentos de modelo explícitos. O compilador deduz esses argumentos dos tipos dos argumentos da função:

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

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

declarações de variável automática

Uma declaração variável pode usar a palavra-chave no lugar do tipo. O compilador deduz o tipo do inicializador da variável, seguindo as mesmas regras da dedução do argumento do modelo de função com o mesmo inicializador (desde que você não use chaves em vez de parênteses). pode ser qualificado com , e pode ser usado como parte de um ponteiro ou tipo de referência, mas não pode ser usado como um argumento de modelo. Uma variante rara desta sintaxe usa em vez de , nesse caso o tipo deduzido é o resultado da aplicação [do_declótipo](#) ao inicializador. `auto`

```
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 an int, not a std::initializer_list<int>
```

`autoconstdecltype(auto)`

Dedução do tipo de retorno da função

`auto (e)` também pode ser usado no lugar de um tipo de retorno de função. O compilador deduz o tipo de retorno das declarações no corpo de função, seguindo as mesmas regras das declarações variáveis: Os tipos de retorno de [expressão lambda](#) podem ser deduzidos da mesma forma, mas isso é desencadeado por omitir o tipo de retorno, e não por um explícito . Confusamente, a sintaxe [do tipo de retorno](#) para funções também usa na posição tipo de retorno, mas isso não depende da dedução do tipo; é apenas uma sintaxe alternativa para um tipo de retorno explícito. `decltype(auto) return`

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

`autoauto`

Lambdas genéricas

Uma expressão lambda pode usar a palavra-chave no lugar de um ou mais de seus tipos de parâmetros. Isso faz com que o operador de chamada da lambda seja um modelo de função em vez de uma função comum, com um parâmetro de modelo separado para cada parâmetro de função: `autoauto`

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

Lambda init captura

As capturas de Lambda podem ter inicializadores explícitos, que podem ser usados para declarar variáveis totalmente novas em vez de apenas capturar as existentes: Esta sintaxe não permite que o tipo seja especificado; em vez disso, é deduzido usando as regras para variáveis.

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

auto

Dedução de argumento do modelo de classe

Veja [abaixo](#).

Amarras estruturadas

Ao declarar uma tupla, estrutura ou matriz usando, você pode especificar nomes para os elementos individuais em vez de um nome para todo o objeto; esses nomes são chamados de "vinculações estruturadas", e toda a declaração é chamada de "declaração vinculante estruturada". Esta sintaxe não fornece nenhuma maneira de especificar o tipo de objeto de fechamento ou os nomes individuais: O também pode ser qualificado com , e , mas note que essas qualificações tecnicamente se aplicam à tupla/struct/array anônimo, em vez das vinculações individuais. As regras que determinam os tipos das vinculações são bastante complexas; os resultados tendem a não ser surpreendentes, exceto que os tipos de vinculação normalmente não serão referências mesmo que a declaração declare uma referência (mas eles geralmente se comportam como referências de qualquer maneira).

auto

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

autoconst&&

(Esses resumos omitem muitos detalhes e ressalvas; consulte os links para obter mais informações.)

Pros:

- Os nomes do tipo C++ podem ser longos e complicados, especialmente quando envolvem modelos ou namespaces.
- Quando um nome do tipo C++ é repetido dentro de uma única declaração ou uma pequena região de código, a repetição pode não estar auxiliando a legibilidade.
- Às vezes é mais seguro deixar o tipo ser deduzido, pois evita a possibilidade de cópias não intencionais ou conversões de tipo.

Cons:

O código C++ geralmente é mais claro quando os tipos são explícitos, especialmente quando a dedução do tipo dependeria de informações de partes distantes do código. Em expressões como:

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

pode não ser óbvio quais são os tipos resultantes se o tipo de não é muito conhecido, ou se foi declarado muitas linhas anteriormente.

Os programadores têm que entender quando a dedução do tipo vai ou não produzir um tipo de referência, ou eles receberão cópias quando não quiserem.

Se um tipo deduzido for usado como parte de uma interface, então um programador pode alterar seu tipo, ao mesmo tempo em que pretende alterar seu valor, levando a uma mudança de API mais radical do que o pretendido.

Decision:

A regra fundamental é: usar a dedução do tipo apenas para tornar o código mais claro ou seguro, e não usá-lo apenas para evitar o inconveniente de escrever um tipo explícito. Ao julgar se o código é mais claro, tenha em mente que seus leitores não estão necessariamente em sua equipe ou familiarizados com seu projeto, de modo que tipos que você e sua experiência de revisor como desordem desnecessária muitas vezes fornecerão informações úteis para outros. Por exemplo, você pode assumir que o tipo de retorno é óbvio, mas o tipo de retorno provavelmente não é `make_unique<Foo>()`

Esses princípios se aplicam a todas as formas de dedução do tipo, mas os detalhes variam, conforme descrito nas seções a seguir.

Dedução do argumento do modelo de função

Dedução do argumento do modelo de função é quase sempre OK. A dedução do tipo é a maneira padrão esperada de interagir com modelos de função, porque permite que modelos de função ajam como conjuntos infinitos de sobrecargas de função comuns. Consequentemente, os modelos de função são quase sempre projetados para que a dedução do argumento do modelo seja clara e segura ou não seja compilada.

Dedução do tipo variável local

Para variáveis locais, você pode usar a dedução do tipo para tornar o código mais claro eliminando informações de tipo que são óbvias ou irrelevantes, para que o leitor possa se concentrar nas partes significativas do código:

```
std::unique_ptr<WidgetWithBellsAndWhistles> widget_ptr =
    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};
```

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

Os tipos às vezes contêm uma mistura de informações úteis e caldeiras, como no exemplo acima: é óbvio que o tipo é um iterador, e em muitos contextos o tipo de recipiente e até mesmo o tipo de chave não são relevantes, mas o tipo de valores é provavelmente útil. Nessas situações, muitas vezes é possível definir variáveis locais com tipos explícitos que transmitem as informações relevantes:`it`

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

Se o tipo for uma instância de modelo, e os parâmetros forem caldeiras, mas o modelo em si for informativo, você pode usar a dedução do argumento do modelo da classe para suprimir a caldeira. No entanto, casos em que isso realmente proporciona um benefício significativo são bastante raros. Observe que a dedução do argumento do modelo de classe também está sujeita a uma [regra de estilo separada](#).

Não use se uma opção mais simples funcionar, porque é um recurso bastante obscuro, por isso tem um alto custo na clareza de código.`decltype(auto)`

Dedução do tipo de retorno

Use dedução do tipo de retorno (para ambas as funções e lambdas) somente se o corpo da função tiver um número muito pequeno de declarações, e muito pouco outro código, pois caso contrário o leitor pode não ser capaz de dizer rapidamente qual é o tipo de retorno. Além disso, use-o somente se a função ou lambda tiver um escopo muito estreito, pois funções com tipos de retorno deduzidos não definem limites de abstração: a implementação é a interface. Em particular, as funções públicas em arquivos de cabeçalho quase nunca deveriam ter deduzido tipos de retorno.`return`

Dedução do tipo parâmetro

auto os tipos de parâmetros para lambdas devem ser usados com cautela, pois o tipo real é determinado pelo código que chama a lambda, e não pela definição da lambda. Consequentemente, um tipo explícito será quase sempre mais claro, a menos que a lambda seja explicitamente chamada muito perto de onde está definida (para que o leitor possa facilmente ver ambos), ou a lambda é passada para uma interface tão conhecida que é óbvio quais argumentos ele eventualmente será chamado (por exemplo, o exemplo acima).`std::sort`

Lambda init captura

As capturas init são cobertas por uma [regra de estilo mais específica](#), que substitui em grande parte as regras gerais para dedução de tipo.

Amarrações estruturadas

Ao contrário de outras formas de dedução de tipo, as vinculações estruturadas podem realmente dar ao leitor informações adicionais, dando nomes significativos aos elementos de um objeto maior. Isso significa que uma declaração vinculante estruturada pode proporcionar uma melhoria de legibilidade líquida em relação a um tipo explícito, mesmo nos casos em que não o faria. As ligações estruturadas são especialmente benéficas quando o objeto é um par ou tupla (como no exemplo acima), porque eles não têm nomes de campo significativos para começar, mas note que você geralmente [não deve usar pares ou tuplas](#) a menos que uma API pré-existente como o `força a fazê-lo.autoinsertinsert`

Se o objeto que está sendo vinculado é uma estrutura, às vezes pode ser útil fornecer nomes mais específicos para seu uso, mas tenha em mente que isso também pode significar que os nomes são menos reconhecíveis para o leitor do que os nomes de campo. Recomendamos o uso de um comentário para indicar o nome do campo subjacente, se ele não corresponder ao nome da vinculação, usando a mesma sintaxe que para comentários de parâmetro de função:

```
auto /*field name1=*/ bound_name1 /*field name2=*/ bound_name2 =
```

```
auto l / field_name1 / domain_name1 / field_name2 / domain_name2 - ...
```

Como com os comentários de parâmetros de função, isso pode permitir que as ferramentas detectem se você errar a ordem dos campos.

⇒ Dedução do argumento do modelo de classe

Use a dedução do argumento do modelo de classe apenas com modelos que optaram explicitamente por apoiá-lo.

Definition:

A [dedução do argumento do modelo de classe](#) (muitas vezes abreviada "CTAD") ocorre quando uma variável é declarada com um tipo que nomeia um modelo, e a lista de argumentos do modelo não é fornecida (nem mesmo suportes de ângulo vazios):

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

O compilador deduz os argumentos do inicializador usando os "guias de dedução" do modelo, que podem ser explícitos ou implícitos.

Guias de dedução explícitos parecem declarações de função com tipos de retorno de arrasto, exceto que não há liderança , e o nome da função é o nome do modelo. Por exemplo, o exemplo acima se baseia neste guia de dedução para:`auto std::array`

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

Construtores em um modelo primário (em oposição a uma especialização de modelo) também definem implicitamente guias de dedução.

Quando você declara uma variável que depende do CTAD, o compilador seleciona um guia de dedução usando as regras de resolução de sobrecarga de construtor, e o tipo de retorno desse guia torna-se o tipo da variável.

Pros:

CtAD às vezes pode permitir que você omita caldeiras a partir de seu código.

Cons:

Os guias implícitos de dedução gerados a partir de construtores podem ter comportamento indesejável, ou estar totalmente incorretos. Isso é particularmente problemático para os construtores escritos antes da 17/17 do CTAD, pois os autores desses construtores não tinham como saber sobre (muito menos fixação) quaisquer problemas que seus construtores causariam para ctad. Além disso, adicionar guias de dedução explícitos para corrigir esses problemas pode quebrar qualquer código existente que dependa dos guias implícitos de dedução.

O CTAD também sofre de muitas das mesmas desvantagens que, pois ambos são mecanismos para deduzir todo ou parte do tipo de uma variável do seu inicializador. A CTAD dá ao leitor mais informações do que , mas também não dá ao leitor uma sugestão óbvia de que as informações foram omitidas.`auto auto`

Decision:

Não use CTAD com um determinado modelo, a menos que os mantenedores do modelo tenham optado por apoiar o uso do CTAD fornecendo pelo menos um guia de dedução explícito (todos os modelos no namespace também são presumidos ter optado). Isso deve ser aplicado com um aviso de compilador se disponível.`std::auto`

Os usos do CTAD também devem seguir as regras gerais sobre [dedução do tipo](#).

⇒ Inicializadores Designados

Use inicializadores designados apenas em seu formulário compatível com C++20.

Definition:

[Os inicializadores designados](#) são uma sintaxe que permite inicializar um agregado ("struct simples e antigo") nomeando seus campos explicitamente:

```
struct Point {
    float x = 0.0;
    float y = 0.0;
    float z = 0.0;
};

Point p = {
    .x = 1.0,
    .y = 2.0,
    // z will be 0.0
};
```

Os campos explicitamente listados serão iniciados conforme especificado, e outros serão iniciados da mesma forma que estariam em uma expressão de inicialização agregada tradicional como `.Point{1.0, 2.0}`

Pros:

Os inicializadores designados podem tornar expressões agregadas convenientes e altamente legíveis, especialmente para estruturas com pedidos menos simples de campos do que o exemplo acima.`Point`

Cons:

Embora os iniciadores designados tenham sido parte do padrão C e suportados por compiladores C++ como uma extensão, apenas recentemente eles entraram no padrão C++, sendo adicionados como parte do C++20.

As regras da norma C++ são mais rigorosas do que em extensões C e compiladores, exigindo que os inicializadores designados apareçam na mesma ordem que os campos aparecem na definição de estrutura. Então, no exemplo acima, é legal de acordo com C++20 inicializar e depois , mas não é depois `.xzyx`

Decision:

Use inicializadores designados apenas na forma compatível com a norma C++20: com inicializadores na mesma ordem que os campos correspondentes aparecem na definição de estrutura.

⇒ Expressões lambda

Use expressões lambda quando apropriado. Prefira capturas explícitas quando a lambda escapará do escopo atual.

Definition:

Expressões lambda são uma maneira concisa de criar objetos de função anônimas. Muitas vezes são úteis ao passar funções como argumentos. Por exemplo:

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

Eles ainda permitem capturar variáveis do escopo de entrada, seja explicitamente pelo nome, ou implicitamente usando uma captura padrão. Capturas explícitas exigem que cada variável seja listada, como um valor ou captura de referência:

```
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;
});
```

Capturas padrão capturam implicitamente qualquer variável referenciada no corpo lambda, inclusive se algum membro for usado:`this`

```
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];
});
```

Uma captura variável também pode ter um inicializador explícito, que pode ser usado para capturar variáveis somente para movimento por valor, ou para outras situações não manipuladas por referências ordinárias ou capturas de valor:

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

Tais capturas (muitas vezes chamadas de "capturas init" ou "capturas de lambda generalizadas") não precisam realmente "capturar" nada do escopo de inc dentro, ou mesmo ter um nome do escopo de inc dentro; esta sintaxe é uma maneira totalmente geral de definir membros de um objeto lambda:

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

O tipo de captura com um inicializador é deduzido usando as mesmas regras que `.auto`

Pros:

- Lambdas são muito mais concisas do que outras formas de definir objetos de função a serem passados para algoritmos STL, o que pode ser uma melhoria de legibilidade.
- O uso adequado de capturas padrão pode remover a redundância e destacar exceções importantes do padrão.

- Lambdas, e pode ser usado em combinação como um mecanismo de retorno de chamada de propósito geral; eles facilitam a escrita de funções que tomam funções vinculadas como argumentos.`std::function``std::bind`

Cons:

- A captura variável em lambdas pode ser uma fonte de bugs de ponteiro pendurado, particularmente se uma lambda escapar do escopo atual.
- Capturas padrão por valor podem ser enganosas porque não impedem bugs de ponteiro pendurados. Capturar um ponteiro por valor não causa uma cópia profunda, por isso muitas vezes tem os mesmos problemas de vida que a captura por referência. Isso é especialmente confuso ao capturar por valor, já que o uso de é muitas vezes implícito.`this`
- Capturas realmente declaram novas variáveis (se as capturas têm ou não inicializadores), mas não se parecem nada com qualquer outra sintaxe de declaração variável em C++. Em particular, não há lugar para o tipo da variável, ou mesmo um espaço reservado (embora capturas init podem indicá-la indiretamente, por exemplo, com um gesso). Isso pode dificultar até mesmo reconhecê-los como declarações.`auto`
- As capturas init dependem inherentemente da [dedução do tipo](#), e sofrem de muitas das mesmas desvantagens que, com o problema adicional de que a sintaxe nem sequer deixa o leitor que a dedução está ocorrendo.`auto`
- É possível usar lambdas para sair do controle; funções anônimas aninhadas muito longas podem tornar o código mais difícil de entender.

Decision:

- Use expressões lambda quando apropriado, com formatação conforme descrito [abaixo](#).
- Prefira capturas explícitas se a lambda pode escapar do escopo atual. Por exemplo, em vez de: prefira escrever:

```
{
    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.
```

```
{
    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.
```

- Use a captura padrão por referência () somente quando a vida útil da lambda for obviamente mais curta do que qualquer captura em potencial. `[&]`
- Use a captura padrão por valor () apenas como um meio de vincular algumas variáveis para uma lambda curta, onde o conjunto de variáveis capturadas é óbvio rapidamente, e que não resulta em captura implícita. (Isso significa que uma lambda que aparece em uma função de membro de classe não estática e se refere a membros de classe não estática em seu corpo deve capturar explicitamente ou via .) Prefira não escrever lambdas longas ou complexas com captura padrão por valor. `[=]this``[&]`
- Use capturas apenas para capturar variáveis do escopo de entrada. Não use capturas com inicializadores para introduzir novos nomes ou para alterar substancialmente o significado de um nome existente. Em vez disso, declare uma nova variável da maneira convencional e, em seguida, capture-a, ou evite a taquigrafia lambda e defina um objeto de função explicitamente.
- Consulte a seção sobre [dedução de tipo](#) para orientação sobre a especificação dos tipos de parâmetro e devolução.

⇒ Metaprogramação de modelos

Evite uma programação de modelos complicada.

Definition:

A metaprogramação do modelo refere-se a uma família de técnicas que exploram o fato de que o mecanismo de instânciação do modelo C++ é Turing completo e pode ser usado para executar uma computação arbitrária de tempo de compilação no domínio do tipo.

Pros:

A metaprogramação do modelo permite interfaces extremamente flexíveis que são tipo segura e de alto desempenho. Instalações como [GoogleTest](#), , , , e [Boost.Spirit](#) seria impossível sem ele.`std::tuple``std::function`

Cons:

As técnicas usadas na metaprogramação de modelos são muitas vezes obscuras para qualquer um, exceto para especialistas em idiomas. O código que usa modelos de maneiras complicadas é muitas vezes ilegível, e é difícil de depurar ou manter.

A metaprogramação do modelo muitas vezes leva a mensagens de erro de tempo de compilação extremamente pobres: mesmo que uma interface seja simples, os complicados detalhes de implementação tornam-se visíveis quando o usuário faz algo errado.

A metaprogramação do modelo interfere na refatorização em larga escala, dificultando o trabalho de refatorar ferramentas. Primeiro, o código de modelo é expandido em vários contextos, e é difícil verificar que a transformação faz sentido em todos eles. Em segundo lugar, algumas ferramentas de refatoração trabalham com um AST que representa apenas a estrutura do código após a expansão do modelo. Pode ser difícil trabalhar automaticamente de volta para a construção de origem original que precisa ser reescrita.

Decision:

A metaprogramação de modelos às vezes permite interfaces mais limpas e fáceis de usar do que seria possível sem ele, mas também muitas vezes é uma tentação de ser excessivamente inteligente. É melhor usado em um pequeno número de componentes de baixo nível onde a carga de manutenção extra é espalhada por um grande número de usos.

Pense duas vezes antes de usar metaprogramação de modelo ou outras técnicas de modelo complicadas; pense se o membro médio da sua equipe será capaz de entender bem o seu código o suficiente para mantê-lo depois de mudar para outro projeto, ou se um programador não-C++ ou alguém navegando casualmente na base de código será capaz de entender as mensagens de erro ou rastrear o fluxo de uma função que eles querem chamar. Se você está usando instâncias de modelo recursivo ou listas de tipos ou metafuncionais ou modelos de expressão, ou confiando no SFINAE ou no truque para detectar a resolução de sobrecarga de função, então há uma boa chance de você ter ido longe demais.sizeof

Se você usar metaprogramação de modelo, você deve esperar para colocar um esforço considerável para minimizar e isolar a complexidade. Você deve ocultar a metaprogramação como um detalhe de implementação sempre que possível, para que os cabeçalhos voltados para o usuário sejam legíveis, e você deve ter certeza de que o código complicado é especialmente bem comentado. Você deve documentar cuidadosamente como o código é usado, e você deve dizer algo sobre como o código "gerado" se parece. Preste atenção extra às mensagens de erro que o compilador emite quando os usuários cometem erros. As mensagens de erro fazem parte da interface do usuário e seu código deve ser ajustado conforme necessário para que as mensagens de erro sejam comprehensíveis e acionáveis do ponto de vista do usuário.

⇒ Impulsionar

Use apenas bibliotecas aprovadas da coleção da biblioteca Boost.

Definition:

A [coleção de bibliotecas Boost](#) é uma coleção popular de bibliotecas C++ de código aberto, revisadas por pares.

Pros:

O código boost é geralmente de alta qualidade, é amplamente portável e preenche muitas lacunas importantes na biblioteca padrão C++, como traços de tipo e aglutinantes melhores.

Cons:

Algumas bibliotecas Boost incentivam práticas de codificação que podem dificultar a legibilidade, como metaprogramação e outras técnicas avançadas de modelo, e um estilo de programação excessivamente "funcional".

Decision:

A fim de manter um alto nível de legibilidade para todos os contribuintes que podem ler e manter o código, só permitimos um subconjunto aprovado de recursos Boost. Atualmente, as seguintes bibliotecas são permitidas:

- [Chame traços](#) de boost/call_traits.hpp
- [Par comprimido](#) de boost/compressed_pair.hpp
- [A Boost Graph Library \(BGL\)](#) de , exceto serialização () e algoritmos paralelos/distribuídos e estruturas de dados (e).boost/graph/adj_list_serialize.hpp boost/graph/parallel/* boost/graph/distributed/*
- [Mapa de Propriedades](#) de , exceto mapas de propriedade paralela/distribuída ().boost/property_map/parallel/*
- [Iterador](#) de boost/iterator
- A parte do [Polígono](#) que lida com a construção do diagrama Voronoi e não depende do resto do Polígono: , e boost/polygon/voronoi_builder.hpp boost/polygon/voronoi_diagram.hpp boost/polygon/voronoi_geometry_type.hpp
- [Bimap](#) de boost/bimap
- [Distribuições e Funções Estatísticas](#) de boost/math/distributions
- [Funções Especiais](#) de boost/math/special_functions
- [Funções de localização de raízes](#) de boost/math/tools
- [Multiíndice](#) de boost/multi_index
- [Monte](#) de boost/heap
- Os recipientes planos do [Container](#): , e boost/container/flat_map boost/container/flat_set
- [Intrusivo](#) de boost/intrusive
- [A biblioteca boost/sort](#).
- [Pré-processador](#) de .boost/preprocessor

Estamos ativamente considerando adicionar outros recursos do Boost à lista, para que esta lista possa ser expandida no futuro.

⇒ Outros recursos C++

Assim como o [Boost](#), algumas extensões C++ modernas incentivam práticas de codificação que dificultam a legibilidade — por exemplo, removendo redundância verificada (como nomes de tipo) que podem ser úteis aos leitores ou incentivando a metaprogramação de modelos. Outras extensões duplicam a funcionalidade disponível através de mecanismos existentes, o que pode levar a custos de confusão e conversão.

Decision:

Além do que está descrito no resto do guia de estilo, os seguintes recursos C++ podem não ser usados:

- Compilar números racionais (), devido a preocupações de que esteja ligado a um estilo de interface mais pesado para modelos.[`<ratio>`](#)
- Os cabeçalhos, porque muitos compiladores não suportam esses recursos de forma confiável.[`<cfev><fev.h>`](#)
- O cabeçalho, que não tem suporte suficiente para testes, e sofre de vulnerabilidades de segurança inerentes.[`<filesystem>`](#)

↔ Extensões não padronizadas

Extensões não padronizadas para C++ não podem ser usadas a menos que seja especificada de outra forma.

Definition:

Os compiladores suportam várias extensões que não fazem parte do C++ padrão. Tais extensões incluem as funções intrínsecas do GCC, tais como, montagem inline, , expressões de declaração composta (por exemplo, , matrizes de comprimento variável e , e o "[Elvis Operator](#)".[`_attribute__builtin_prefetch_COUNTER__PRETTY_FUNCTION_`](#)`foo = ({ int x; Bar(&x); x })alloca()a?:b`

Pros:

- Extensões não padronizadas podem fornecer recursos úteis que não existem no C++ padrão.
- Orientações de desempenho importantes para o compilador só podem ser especificadas usando extensões.

Cons:

- Extensões não padronizadas não funcionam em todos os compiladores. O uso de extensões não padronizadas reduz a portabilidade do código.
- Mesmo que sejam suportados em todos os compiladores-alvo, as extensões muitas vezes não são bem especificadas, e pode haver diferenças de comportamento sutis entre compiladores.
- Extensões não padronizadas adicionam aos recursos de idioma que um leitor deve saber para entender o código.

Decision:

Não use extensões não padronizadas. Você pode usar invólucros de portabilidade que são implementados usando extensões não padronizadas, desde que esses invólucros sejam fornecidos por um cabeçalho de portabilidade designado em todo o projeto.

↔ Aliases

Os pseudônimos públicos são para o benefício do usuário de uma API e devem ser claramente documentados.

Definition:

Existem várias maneiras de criar nomes que são pseudônimos de outras entidades:

```
typedef Foo Bar;
using Bar = Foo;
using other_namespace::Foo;
```

No novo código, é preferível, pois fornece uma sintaxe mais consistente com o resto do C++ e funciona com modelos.[`usingtypedef`](#)

Como outras declarações, os pseudônimos declarados em um arquivo de cabeçalho fazem parte da API pública desse cabeçalho, a menos que estejam em uma definição de função, na parte privada de uma classe ou em um namespace interno explicitamente marcado. Codinomes nessas áreas ou em arquivos .cc são detalhes de implementação (porque o código do cliente não pode se referir a eles) e não são restritos por essa regra.

Pros:

- Codinomes podem melhorar a legibilidade simplificando um nome longo ou complicado.
- Os pseudônimos podem reduzir a duplicação nomeando em um lugar um tipo usado repetidamente em uma API, o que pode facilitar a alteração do tipo mais tarde.

Cons:

- Quando colocado em um cabeçalho onde o código do cliente pode se referir a eles, os aliases aumentam o número de entidades na API desse cabeçalho, aumentando sua complexidade.
- Os clientes podem facilmente confiar em detalhes não intencionais de pseudônimos públicos, dificultando as mudanças.
- Pode ser tentador criar um pseudônimo público que se destina apenas ao uso na implementação, sem considerar seu impacto na API, ou na manutenção.
- Codinomes podem criar risco de colisões de nomes

- Codinomes podem reduzir a legibilidade dando a uma construção familiar um nome desconhecido
- Codinomes de tipo podem criar um contrato de API pouco claro: não está claro se o alias é garantido ser idêntico ao tipo que ele aliases, ter a mesma API, ou apenas ser utilizável de maneiras estreitas especificadas

Decision:

Não coloque um pseudônimo em sua API pública apenas para salvar a digitação na implementação; fazê-lo apenas se você pretende que ele seja usado por seus clientes.

Ao definir um pseudônimo público, documente a intenção do novo nome, incluindo se é garantido que ele seja sempre o mesmo do tipo a que é atualmente aliased, ou se uma compatibilidade mais limitada é pretendida. Isso permite que o usuário saiba se pode tratar os tipos como substituíveis ou se regras mais específicas devem ser seguidas, e pode ajudar a implementação a manter algum grau de liberdade para alterar o pseudônimo.

Não coloque codinomes namespace em sua API pública. (Veja também [Namespaces](#)).

Por exemplo, esses codinomes documentam como eles são destinados a serem usados no código do cliente:

```
namespace mynamespace {
// Used to store field measurements. DataPoint may change from Bar* to some internal type.
// 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>, DataPointComparator>;
} // namespace mynamespace
```

Esses pseudônimos não documentam o uso pretendido, e metade deles não são destinados ao uso do cliente:

```
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
```

No entanto, os pseudônimos de conveniência locais são bons em definições de função, seções privadas de classes, espaços de nome internos explicitamente marcados e em arquivos .cc:

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

☞ Linguagem Inclusiva

Em todos os códigos, incluindo nomeação e comentários, use linguagem inclusiva e evite termos que outros programadores possam achar desrespeitoso ou ofensivo (como "mestre" e "escravo", "lista negra" e "lista branca", ou "linha vermelha"), mesmo que os termos também tenham um significado ostensivamente neutro. Da mesma forma, use linguagem neutra em termos de gênero, a menos que você esteja se referindo a uma pessoa específica (e usando seus pronomes). Por exemplo, use "eles"/"eles"/"seus" para pessoas de gênero não especificado ([mesmo quando singular](#)), e "ele"/"its" para softwares, computadores e outras coisas que não são pessoas.

☞ Nomeação

As regras de consistência mais importantes são aquelas que governam a nomeação. O estilo de um nome imediatamente nos informa que tipo de coisa a entidade nomeada é: um tipo, uma variável, uma função, uma constante, uma macro, etc., sem exigir que procuremos a declaração dessa entidade. O motor que combina padrões em nossos cérebros depende muito dessas regras de nomeação.

As regras de nomeação são bastante arbitrárias, mas achamos que a consistência é mais importante do que as preferências individuais nesta área, portanto, independentemente de você as achar sensatas ou não, as regras são as regras.

☞ Regras gerais de nomeação

Otimizar para legibilidade usando nomes que seriam claros até mesmo para pessoas de uma equipe diferente.

Use nomes que descrevam o propósito ou a intenção do objeto. Não se preocupe em economizar espaço horizontal, pois é muito mais importante tornar seu código imediatamente compreensível por um novo leitor. Minimize o uso de abreviaturas que provavelmente seriam desconhecidas para alguém fora do seu projeto (especialmente siglas e inicialismos). Não abreviar excluindo letras dentro de uma palavra. Como regra geral, uma abreviação provavelmente

está OK se estiver listada na Wikipédia. De um modo geral, a descriptiva deve ser proporcional ao escopo de visibilidade do nome. Por exemplo, pode ser um bom nome dentro de uma função de 5 linhas, mas dentro do escopo de uma classe, é provavelmente muito vago.

```
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  
};
```

```
class MyClass {  
public:  
    int CountFooErrors(const std::vector<Foo>& foos) {  
        int total_number_of_foo_errors = 0; // Overly verbose given limited scope and context  
        for (int foo_index = 0; foo_index < foos.size(); ++foo_index) { // Use idiomatic `i`  
            ...  
            ++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  
};
```

Observe que certas abreviaturas universalmente conhecidas são OK, como para uma variável de iteração e para um parâmetro de modelo.

Para efeitos das regras de nomeação abaixo, uma "palavra" é qualquer coisa que você escreveria em inglês sem espaços internos. Isso inclui abreviaturas, como siglas e inicialismos. Para nomes escritos em maiúsculas (também algumas vezes referidos como "[caso camel](#)" ou "[caso Pascal](#)"), em que a primeira letra de cada palavra é capitalizada, prefira capitalizar abreviaturas como palavras simples, por exemplo, em vez de `.StartRpc()`/`StartRPC()`

Os parâmetros do modelo devem seguir o estilo de nomeação para sua categoria: parâmetros de modelo de tipo devem seguir as regras para [nomes de tipo](#), e parâmetros de modelo não-tipo devem seguir as regras para [nomes de variáveis](#).

☞ Nomes de arquivos

Os nomes dos arquivos devem ser todos minúsculos e podem incluir sublinhados () ou traços (). Siga a convenção que seu projeto usa. Se não houver um padrão local consistente a seguir, prefira `_.-`

Exemplos de nomes de arquivos aceitáveis:

- `my_useful_class.cc`
- `my-useful-class.cc`
- `myusefulclass.cc`
- `myusefulclass_test.cc` // `_unittest` and `_regtest` are deprecated.

Os arquivos C++ devem terminar e os arquivos de cabeçalho devem terminar . Os arquivos que dependem de serem incluídos texuicamente em pontos específicos devem terminar (veja também a seção sobre [cabeçalhos independentes](#)).cc.h.inc

Não use nomes de arquivos que já existem em , tais como `./usr/include/db.h`

Em geral, torne seus nomes de arquivos muito específicos. Por exemplo, use em vez de . Um caso muito comum é ter um par de arquivos chamados, por exemplo, e , definindo uma classe chamada `.http_server_logs.hlogs.hfoo_bar.hfoo_bar.ccFooBar`

☞ Nomes de tipo

Os nomes dos tipos começam com uma letra maiúscula e têm uma letra maiúscula para cada nova palavra, sem sublinhar: `.MyExcitingClassMyExcitingEnum`

Os nomes de todos os tipos — classes, estruturas, codinomes, enums e parâmetros de modelo de tipo — têm a mesma convenção de nomeação. Os nomes de digitar devem começar com uma letra maiúscula e ter uma letra maiúscula para

cada nova palavra. Sem sublinhados. 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 variáveis

Os nomes das variáveis (incluindo parâmetros de função) e dos membros dos dados são todos minúsculos, com sublinhados entre as palavras. Os membros de dados das classes (mas não as estruturas) também têm sublinhados. Por exemplo:`a_local_variablea_struct_data_membera_class_data_member_`

Nomes de variável comuns

Por exemplo:

```
std::string table_name; // OK - lowercase with underscore.
```

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

Membros de dados de classe

Os membros dos dados das classes, tanto estáticas quanto não estáticas, são nomeados como variáveis comuns não-membros, mas com um sublinhado.

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

Membros de dados de estrutura

Os membros de dados de estruturas, tanto estáticas quanto não estáticas, são nomeados como variáveis não-membros comuns. Eles não têm os sublinhados que os membros de dados nas aulas têm.

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

Consulte [Structs vs. Classes](#) para uma discussão sobre quando usar uma estrutura versus uma classe.

☞ Nomes constantes

As variáveis declaradas `constexpr` ou `const`, e cujo valor é fixado durante a duração do programa, são nomeadas com um "k" líder seguido de caso misto. Os sublinhados podem ser usados como separadores nos raros casos em que a capitalização não pode ser usada para separação. Por exemplo:

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

Todas essas variáveis com duração de armazenamento estática (ou seja, estáticas e globais, consulte [Duração do armazenamento](#) para detalhes) devem ser nomeadas desta forma. Esta convenção é opcional para variáveis de outras classes de armazenamento, por exemplo, variáveis automáticas, caso contrário as regras habituais de nomeação variável se aplicam.

⇒ Nomes de função

Funções regulares têm caso misto; acessórios e mutadores podem ser nomeados como variáveis.

Normalmente, as funções devem começar com uma letra maiúscula e ter uma letra maiúscula para cada nova palavra.

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

(A mesma regra de nomeação se aplica às constantes de escopo de classe e namespace que são expostas como parte de uma API e que se destinam a parecer funções, porque o fato de serem objetos em vez de funções é um detalhe de implementação sem importância.)

Acessórios e mutadores (funções de get e set) podem ser nomeados como variáveis. Estas muitas vezes correspondem a variáveis reais de membros, mas isso não é necessário. Por exemplo,

```
e.int count()void set_count(int count)
```

⇒ Nomes do namespace

Os nomes do namespace são todos minúsculas, com palavras separadas por sublinhados. Os nomes de namespace de alto nível são baseados no nome do projeto . Evite colisões entre espaços de nome aninhados e espaços de nome de alto nível bem conhecidos.

O nome de um namespace de alto nível geralmente deve ser o nome do projeto ou equipe cujo código está contido nesse namespace. O código nesse namespace geralmente deve estar em um diretório cujo nome de base corresponde ao nome do namespace (ou em subdiretórios).

Tenha em mente que a [regra contra nomes abreviados](#) se aplica aos namespaces tanto quanto aos nomes de variáveis. O código dentro do namespace raramente precisa mencionar o nome do namespace, então geralmente não há necessidade particular de abreviação de qualquer maneira.

Evite espaços de nome aninhados que combinem com espaços de nome de alto nível bem conhecidos. Colisões entre nomes de namespace podem levar a surpreendentes quebras de compilação por causa das regras de busca de nomes. Em particular, não crie espaços de nome aninhados. Prefira identificadores de projeto exclusivos (), em vez de nomes propensos a colisão como . Evite também espaços de nomes de ninhos excessivamente profundos ([TotW #130](#)).stdwebsearch::indexwebsearch::index_utilwebsearch::util

Para namespaces, tenha cuidado com o outro código que está sendo adicionado ao mesmo namespace causando uma colisão (ajudantes internos dentro de uma equipe tendem a estar relacionados e podem levar a colisões). Em tal situação, usar o nome do arquivo para fazer um nome interno exclusivo é útil (para uso em).internalinternalwebsearch::index::frobber_internalfrobber.h

⇒ Nomes do Enumerador

Os enumeradores (tanto para enums escopo quanto não escópio) devem ser [nomeados como constantes](#), não como [macros](#). Ou seja, não use.kEnumNameENUM_NAME

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

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

Até janeiro de 2009, o estilo era nomear valores de enum como [macros](#). Isso causou problemas com colisões de nomes entre valores de enum e macros. Assim, a mudança para preferir o nome constante foi posta em prática. O novo código deve usar nomeação em estilo constante.

⇒ Nomes de macro

Você não vai realmente [definir uma macro](#), vai? Se você fizer isso, eles são assim: .
MY_MACRO_THAT_SCARES_SMALL_CHILDREN_AND_ADULTS_ALIKE

Consulte a [descrição das macros](#): em macros gerais *não* devem ser utilizadas. No entanto, se eles são absolutamente necessários, então eles devem ser nomeados com todas as capitais e sublinhados.

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

⇒ Exceções às regras de nomeação

Se você está nomeando algo que é análogo a uma entidade C ou C++ existente, então você pode seguir o esquema de convenção de nomeação existente.

```
bigopen()
    nome da função, segue a forma de open()
uint
    typedef
bigpos
    struct ou , segue a forma de classpos
sparse_hash_map
    Entidade semelhante ao STL; segue convenções de nomeação STL
LONGLONG_MAX
    uma constante, como em INT_MAX
```

⇒ Comentários

Os comentários são absolutamente vitais para manter nosso código legível. As seguintes regras descrevem o que você deve comentar e onde. Mas lembre-se: embora os comentários sejam muito importantes, o melhor código é auto-documentação. Dar nomes sensíveis a tipos e variáveis é muito melhor do que usar nomes obscuros que você deve explicar através de comentários.

Ao escrever seus comentários, escreva para o seu público: o próximo colaborador que precisará entender o seu código. Seja generoso - o próximo pode ser você!

⇒ Estilo de comentário

Use a sintaxe ou a sintaxe, desde que seja consistente.///* */

Você pode usar a sintaxe ou a sintaxe; no entanto, é *muito* mais comum. Seja coerente com a forma como você comenta e que estilo você usa onde.///* */

⇒ Comentários de arquivos

Inicie cada arquivo com placa de caldeira de licença.

Os comentários do arquivo descrevem o conteúdo de um arquivo. Se um arquivo declarar, implementar ou testar exatamente uma abstração documentada por um comentário no momento da declaração, os comentários de arquivo não são necessários. Todos os outros arquivos devem ter comentários de arquivo.

Editor e Linha do Autor

Todos os arquivos devem conter caldeiras de licença. Escolha a caldeira apropriada para a licença utilizada pelo projeto (por exemplo, Apache 2.0, BSD, LGPL, GPL).

Se você fizer alterações significativas em um arquivo com uma linha autoral, considere excluir a linha autor. Os novos arquivos geralmente não devem conter aviso de direitos autorais ou linha de autor.

Conteúdo do arquivo

Se um declarar múltiplas abstrações, o comentário em nível de arquivo deve descrever amplamente o conteúdo do arquivo e como as abstrações estão relacionadas. Um comentário de 1 ou 2 frases em nível de arquivo pode ser suficiente. A documentação detalhada sobre abstrações individuais pertence a essas abstrações, não ao nível do arquivo..h

Não duplique comentários tanto no . Comentários duplicados divergem..h.cc

⇒ Comentários de Classe

Toda declaração de classe ou estrutura não óbvia deve ter um comentário que descreva para que é e como deve ser usado.

```
// 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 {  
    ...  
};
```

O comentário da classe deve fornecer ao leitor informações suficientes para saber como e quando usar a classe, bem como quaisquer considerações adicionais necessárias para usar corretamente a classe. Documente as suposições de sincronização que a classe faz, se houver. Se uma instância da classe pode ser acessada por vários segmentos, tome cuidado extra para documentar as regras e invariantes em torno do uso multithreaded.

O comentário de classe é muitas vezes um bom lugar para um pequeno exemplo de trecho de código demonstrando um uso simples e focado da classe.

Quando suficientemente separados (por exemplo, e arquivos), os comentários que descrevem o uso da classe devem ir em conjunto com sua definição de interface; comentários sobre a operação e implementação da classe devem acompanhar a implementação dos métodos da classe..h.cc

⇒ Comentários de função

Os comentários de declaração descrevem o uso da função (quando não é óbvio); comentários na definição de uma função descrever operação.

Declarações de Função

Quase todas as declarações de função devem ter comentários imediatamente anteriores que descrevam o que a função faz e como usá-la. Esses comentários só podem ser omitidos se a função for simples e óbvia (por exemplo, acessórios simples para propriedades óbvias da classe). Métodos e funções privadas declaradas em arquivos '.cc' não são isentos. Os comentários de função devem ser escritos com um assunto implícito *desta função* e devem começar com a frase do verbo; por exemplo, "Abre o arquivo", em vez de "Abrir o arquivo". Em geral, esses comentários não descrevem como a função realiza sua tarefa. Em vez disso, isso deve ser deixado para comentários na definição de função.

Tipos de coisas a serem mencionadas nos comentários da declaração de função:

- Quais são as entradas e saídas. Se os nomes dos argumentos de função forem fornecidos em 'backticks', então as ferramentas de indexação de código podem ser capazes de apresentar melhor a documentação.
- Para funções de membro da classe: se o objeto se lembra de argumentos de referência além da duração da chamada do método, e se irá libertá-los ou não.
- Se a função alocar memória que o chamador deve liberar.
- Se algum dos argumentos pode ser um ponteiro nulo.
- Se houver alguma implicação de desempenho de como uma função é usada.
- Se a função for re-entrante. Quais são suas suposições de sincronização?

Aqui está um exemplo:

```
// 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;
```

No entanto, não seja desnecessariamente verboso ou indique o completamente óbvio.

Ao documentar as substituições da função, concentre-se nas especificidades da própria substituição, em vez de repetir o comentário da função substituída. Em muitos desses casos, a substituição não precisa de documentação adicional e, portanto, não é necessário comentar.

Ao comentar construtores e destruidores, lembre-se que a pessoa que lê seu código sabe para que são os construtores e destruidores, então comentários que apenas dizem algo como "destrói esse objeto" não são úteis. Documente o que os construtores fazem com seus argumentos (por exemplo, se eles tomam posse de ponteiros) e o que a limpeza do destruidor faz. Se isso é trivial, basta pular o comentário. É bastante comum que os destruidores não tenham um comentário de cabeçalho.

Definições de função

Se há algo complicado sobre como uma função faz seu trabalho, a definição da função deve ter um comentário explicativo. Por exemplo, no comentário de definição você pode descrever quaisquer truques de codificação que você usa, dar uma visão geral das etapas pelas quais você passa ou explicar por que você escolheu implementar a função da maneira que você fez em vez de usar uma alternativa viável. Por exemplo, você pode mencionar por que ele deve adquirir um bloqueio para a primeira metade da função, mas por que não é necessário para o segundo semestre.

Observe que você *não* deve apenas repetir os comentários dados com a declaração de função, no arquivo ou em qualquer lugar. Não há problema em recapitular brevemente o que a função faz, mas o foco dos comentários deve ser sobre como ele faz isso..h

Comentários variáveis

Em geral, o nome real da variável deve ser descritivo o suficiente para dar uma boa ideia de para que a variável é utilizada. Em certos casos, mais comentários são necessários.

Membros de dados de classe

O propósito de cada membro de dados de classe (também chamado de variável de instância ou variável membro) deve ser claro. Se houver alguma invariante (valores especiais, relações entre membros, requisitos vitalícios) não expressas claramente pelo tipo e nome, devem ser comentadas. No entanto, se o tipo e o nome forem suficientes (), nenhum comentário é necessário.`int num_events_;`

Em particular, adicione comentários para descrever a existência e o significado dos valores sentinelas, como nulo ou -1, quando não são óbvios. Por exemplo:

```
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_;
```

Variáveis Globais

Todas as variáveis globais devem ter um comentário descrevendo o que são, para que são usadas e (se não está claro) por que precisam ser globais. Por exemplo:

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

Comentários de implementação

Em sua implementação, você deve ter comentários em partes complicadas, não óbvias, interessantes ou importantes do seu código.

Comentários Explicativos

Blocos de código complicados ou complicados devem ter comentários antes deles.

⇒ Comentários de argumento de função

Quando o significado de um argumento de função é não-obvious, considere um dos seguintes remédios:

- Se o argumento é uma constante literal, e a mesma constante é usada em chamadas de múltiplas funções de uma maneira que facilmente assume que eles são os mesmos, você deve usar uma constante nomeada para tornar essa restrição explícita, e para garantir que ela se mantenha.
- Considere alterar a assinatura da função para substituir um argumento por um argumento. Isso fará com que os valores do argumento se auto-descrevam.`bool enum`
- Para funções que possuem várias opções de configuração, considere definir uma única classe ou estrutura para manter todas as opções, e passar uma instância disso. Essa abordagem tem várias vantagens. As opções são referenciadas pelo nome no site de chamadas, o que esclarece seu significado. Também reduz a contagem de argumentos de função, o que torna as chamadas de função mais fáceis de ler e escrever. Como um benefício adicional, você não precisa alterar sites de chamadas quando adiciona outra opção.
- Substitua expressões aninhadas grandes ou complexas por variáveis nomeadas.
- Como último recurso, use comentários para esclarecer significados de argumentos no site de chamadas.

Considere o seguinte exemplo:

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

contra:

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

⇒ Contras

Não se diz o óbvio. Em particular, não descreva literalmente o que o código faz, a menos que o comportamento seja não duvidoso para um leitor que entende bem C++. Em vez disso, forneça comentários de nível mais alto que *descrevam por que* o código faz o que ele faz, ou faz o código se auto-descrever.

Compare isso: Com isso: O código auto-descritor não precisa de um comentário. O comentário do exemplo acima seria óbvio:

```
// 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);  
}
```

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

⇒ Pontuação, Ortografia e Gramática

Preste atenção à pontuação, ortografia e gramática; é mais fácil ler comentários bem escritos do que mal escritos.

Os comentários devem ser tão legíveis quanto texto narrativo, com capitalização e pontuação adequadas. Em muitos casos, frases completas são mais legíveis do que fragmentos de sentença. Comentários mais curtos, como comentários no final de uma linha de código, às vezes podem ser menos formais, mas você deve ser consistente com o seu estilo.

Embora possa ser frustrante ter um revisor de código salientando que você está usando uma vírgula quando você deve estar usando um ponto e vírgula, é muito importante que o código fonte mantenha um alto nível de clareza e legibilidade. Pontuação adequada, ortografia e gramática ajudam nesse objetivo.

⇒ TODOS Comentários

Use comentários para códigos temporários, uma solução de curto prazo ou bom o suficiente, mas não perfeito.TODO

TODOs deve incluir a sequência em todas as tampas, seguida pelo nome, endereço de e-mail, ID de bug ou outro identificador da pessoa ou problema com o melhor contexto sobre o problema mencionado pelo . O objetivo principal é ter um consistente que possa ser pesquisado para saber como obter mais detalhes mediante solicitação. A não é um compromisso que a pessoa referenciada vai corrigir o problema. Assim, quando você cria um com um nome, é quase sempre o seu nome que é dado.TODOTODOTODOTODOTODO

```
// TODO(kl@gmail.com): Use a "*" here for concatenation operator.  
// TODO(Zeke) change this to use relations.  
// TODO(bug 12345): remove the "Last visitors" feature.
```

Se o seu for do formulário "Em uma data futura faça alguma coisa" certifique-se de que você inclua uma data muito específica ("Fix até novembro de 2005") ou um evento muito específico ("Remova este código quando todos os clientes podem lidar com respostas XML.").TODO

☞ Formatação

O estilo de codificação e a formatação são bastante arbitrários, mas um projeto é muito mais fácil de seguir se todos usam o mesmo estilo. Os indivíduos podem não concordar com todos os aspectos das regras de formatação, e algumas das regras podem levar algum tempo para se acostumar, mas é importante que todos os colaboradores do projeto sigam as regras de estilo para que todos possam ler e entender o código de todos facilmente.

Para ajudá-lo a formatar o código corretamente, criamos um [arquivo de configurações para emacs](#).

☞ Comprimento da linha

Cada linha de texto em seu código deve ter no máximo 80 caracteres de comprimento.

Reconhecemos que essa regra é controversa, mas tanto código existente já adere a ela, e achamos que a consistência é importante.

Pros:

Aqueles que favorecem essa regra argumentam que é rude forçá-los a redimensionar suas janelas e não há necessidade de mais nada. Algumas pessoas estão acostumadas a ter várias janelas de código lado a lado, e, portanto, não têm espaço para ampliar suas janelas em qualquer caso. As pessoas configuraram seu ambiente de trabalho assumindo uma largura máxima de janela específica, e 80 colunas tem sido o padrão tradicional. Por que mudá-lo?

Cons:

Os defensores da mudança argumentam que uma linha mais ampla pode tornar o código mais legível. O limite de 80 colunas é um retrocesso escondido para mainframes dos anos 1960; equipamentos modernos têm telas largas que podem facilmente mostrar linhas mais longas.

Decision:

80 caracteres é o máximo.

Uma linha pode exceder 80 caracteres se for

- uma linha de comentários que não é viável de dividir sem prejudicar a legibilidade, facilidade de cortar e colar ou vincular automaticamente -- por exemplo, se uma linha contiver um comando de exemplo ou uma URL literal com mais de 80 caracteres.
- um literal de string bruta com conteúdo que excede 80 caracteres. Exceto pelo código de teste, tais literais devem aparecer perto do topo de um arquivo.
- uma declaração de inclusão.
- um [guarda cabeçalho](#)
- uma declaração de uso

☞ Caracteres não-ASCII

Caracteres não ASCII devem ser raros e devem usar a formatação UTF-8.

Você não deve codificar texto voltado para o usuário na fonte, mesmo em inglês, por isso o uso de caracteres não-ASCII deve ser raro. No entanto, em certos casos, é apropriado incluir tais palavras em seu código. Por exemplo, se o seu código analisar arquivos de dados de fontes estrangeiras, pode ser apropriado codificar as strings não-ASCII usadas nesses arquivos de dados como delimitadores. Mais comumente, o código unittest (que não precisa ser localizado) pode conter strings não-ASCII. Nesses casos, você deve usar o UTF-8, já que essa é uma codificação entendida pela maioria das ferramentas capazes de lidar com mais do que apenas ASCII.

A codificação hexax também é OK, e encorajada onde melhora a legibilidade — por exemplo, ou, ainda mais simplesmente, é o caractere espaço unicode de largura zero sem quebra, que seria invisível se incluído na fonte como UTF-8 em linha reta. "\xEF\xBB\xBF"\u8"\uFEFF"

Use o prefixo para garantir que uma sequência literal de sequências de fuga seja codificada como UTF-8. Não o use para strings que contenham caracteres não ASCII codificados como UTF-8, porque isso produzirá saída incorreta se o compilador não interpretar o arquivo de origem como UTF-8. \u8\uXXXX

Você não deve usar e tipos de caracteres, já que são para texto não-UTF-8. Por razões semelhantes, você também não deve usar (a menos que você esteja escrevendo código que interage com a API do Windows, que usa extensivamente).char16_tchar32_twchar_twchar_t

↔ Espaços vs. Guias

Use apenas espaços e 2 espaços de cada vez.

Usamos espaços para recuo. Não use guias em seu código. Você deve configurar seu editor para emitir espaços quando acertar a tecla de guia.

↔ Declarações e Definições de Função

Retorne digitar na mesma linha do nome da função, parâmetros na mesma linha se eles se encaixam. Enrole listas de parâmetros que não se encaixam em uma única linha, pois você envolveria argumentos em uma [chamada de função](#).

As funções são assim:

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

Se você tem muito texto para caber em uma linha:

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

ou se você não pode caber nem mesmo no primeiro parâmetro:

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

Alguns pontos a notar:

- Escolha bons nomes de parâmetros.
- Um nome de parâmetro só pode ser omitido se o parâmetro não for usado na definição da função.
- Se você não puder encaixar o tipo de retorno e o nome da função em uma única linha, rompa entre eles.
- Se você quebrar após o tipo de retorno de uma declaração ou definição de função, não faça recuo.
- O parêntese aberto está sempre na mesma linha do nome da função.
- Nunca há um espaço entre o nome da função e o parêntese aberto.
- Nunca há um espaço entre os parênteses e os parâmetros.
- A chave encaracolada aberta está sempre no final da última linha da declaração de função, não no início da próxima linha.
- A cinta cacheado perto está na última linha por si só ou na mesma linha que a cinta encaracolada aberta.
- Deve haver um espaço entre o parêntese próximo e a cinta encaracolada aberta.
- Todos os parâmetros devem ser alinhados, se possível.
- O recuo padrão é de 2 espaços.
- Os parâmetros embrulhados têm um recuo de 4 espaços.

Parâmetros não usos que são óbvios do contexto podem ser omitidos:

```
class Foo {  
public:  
    Foo(const Foo&) = delete;  
    Foo& operator=(const Foo&) = delete;  
};
```

Parâmetros não usados que podem não ser óbvios devem comentar o nome variável na definição da função:

```

class Shape {
public:
    virtual void Rotate(double radians) = 0;
};

class Circle : public Shape {
public:
    void Rotate(double radians) override;
};

void Circle::Rotate(double /*radians*/) {}

```

```

// Bad - if someone wants to implement later, it's not clear what the
// variable means.
void Circle::Rotate(double) {}

```

Atributos e macros que se expandem aos atributos aparecem no início da declaração ou definição da função, antes do tipo de retorno:

```
ABSL_MUST_USE_RESULT bool IsOk();
```

⇒ Expressões lambda

Formava parâmetros e corpos como para qualquer outra função e capture listas como outras listas separadas por címulas.

Para capturas por referência, não deixe um espaço entre o ampersand (&) e o nome variável.

```
int x = 0;
auto x_plus_n = [&x](int n) -> int { return x + n; }
```

Lambdas curtas podem ser escritas em linha como argumentos de função.

```

std::set<int> to_remove = {7, 8, 9};
std::vector<int> digits = {3, 9, 1, 8, 4, 7, 1};
digits.erase(std::remove_if(digits.begin(), digits.end(), [&to_remove](int i) {
    return to_remove.find(i) != to_remove.end();
}),
digits.end());

```

⇒ Literais de ponto flutuante

Os literais de ponto flutuante devem sempre ter um ponto radix, com dígitos em ambos os lados, mesmo que usem notação exponencial. A legibilidade é melhorada se todos os literal de ponto flutuante tomarem essa forma familiar, pois isso ajuda a garantir que eles não sejam confundidos com literais inteiros, e que a/ da notação exponencial não seja confundida com um dígito hexadecimal. É bom inicializar uma variável de ponto flutuante com um inteiro literal (assumindo que o tipo variável pode representar exatamente esse inteiro), mas note que um número em notação exponencial nunca é um literal inteiro. Ee

```
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;
```

⇒ Chamadas de função

Escreva a chamada tudo em uma única linha, enrole os argumentos no parêntese, ou inicie os argumentos em uma nova linha recuada por quatro espaços e continue naquele recuo de 4 espaços. Na ausência de outras considerações, use o número mínimo de linhas, incluindo a colocação de múltiplos argumentos em cada linha, quando apropriado.

As chamadas de função têm o seguinte formato:

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

Se os argumentos não se encaixam em uma linha, eles devem ser divididos em várias linhas, com cada linha subsequente alinhada com o primeiro argumento. Não adicione espaços após o paren aberto ou antes do paren próximo:

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

Os argumentos podem, opcionalmente, ser colocados em linhas subsequentes com um recuo de quatro espaços:

```
if (...) {
    ...
    if (...) {
        bool result = DoSomething(
            argument1, argument2, // 4 space indent
            argument3, argument4);
    }
}
```

Coloque vários argumentos em uma única linha para reduzir o número de linhas necessárias para chamar uma função, a menos que haja um problema específico de legibilidade. Alguns acham que a formatação com um argumento estritamente um em cada linha é mais legível e simplifica a edição dos argumentos. No entanto, priorizamos para o leitor sobre a facilidade de editar argumentos, e a maioria dos problemas de legibilidade são melhor abordados com as seguintes técnicas.

Se ter vários argumentos em uma única linha diminui a legibilidade devido à complexidade ou natureza confusa das expressões que compõem alguns argumentos, tente criar variáveis que capturem esses argumentos em um nome descritivo:

```
int my_heuristic = scores[x] * y + bases[x];
bool result = DoSomething(my_heuristic, x, y, z);
```

Ou colocar o argumento confuso em sua própria linha com um comentário explicativo:

```
bool result = DoSomething(scores[x] * y + bases[x], // Score heuristic.
                           x, y, z);
```

Se ainda há um caso em que um argumento é significativamente mais legível em sua própria linha, em seguida, coloca-lo em sua própria linha. A decisão deve ser específica do argumento que se torna mais legível do que uma política geral.

Às vezes, os argumentos formam uma estrutura importante para a legibilidade. Nesses casos, sinta-se livre para formatar os argumentos de acordo com essa estrutura:

```
// Transform the widget by a 3x3 matrix.
my_widget.Transform(x1, x2, x3,
                    y1, y2, y3,
                    z1, z2, z3);
```

☞ Formato da lista de inicializadores de cinta

Formata uma lista de inicializadores preparados exatamente como você formataria uma chamada de função em seu lugar.

Se a lista de aparelhos seguir um nome (por exemplo, um tipo ou nome variável), formata como se fossem os parênteses de uma chamada de função com esse nome. Se não houver nome, assuma um nome de comprimento zero.{}{}

```
// Examples of braced init list on a single line.
return {foo, bar};
functioncall({foo, bar});
std::pair<int, int> p{foo, bar};

// When you have to wrap.
SomeFunction(
    {"assume a zero-length name before {}",
     some_other_function_parameter});
SomeType variable{
    some, other, values,
    {"assume a zero-length name before {}",
     SomeOtherType{
         "Very long string requiring the surrounding breaks.",
         some, other, values},
     SomeOtherType{"Slightly shorter string",
                  some, other, values}});
SomeType variable{
    "This is too long to fit all in one line"};
MyType m = { // Here, you could also break before {
    superlongvariablename1,
    superlongvariablename2,
```

```
{short, interior, list},  
{interiorwrappinglist,  
interiorwrappinglist2}};
```

⇒ Condicionais

Em uma declaração, incluindo suas cláusulas e opcionais, coloque um espaço entre o parêntese de abertura e entre o parêntese de fechamento e a cinta encaracolada (se houver), mas sem espaços entre os parênteses e a condição ou inicializador. Se o inicializador opcional estiver presente, coloque um espaço ou uma linha nova após o ponto e vírgula, mas não antes.`if``else` `if``elseif`

```
if(condition) { // Bad - space missing after IF  
if ( condition ) { // Bad - space between the parentheses and the condition  
if (condition){ // Bad - space missing before {  
if(condition){ // Doubly bad  
  
if (int a = f();a == 10) { // Bad - space missing after the semicolon
```

Use chaves para as instruções controladas seguintes, e . Quebre a linha imediatamente após a abertura da cinta, e imediatamente antes da cinta de fechamento. Um subsequente, se houver, aparece na mesma linha da cinta de fechamento anterior, separada por um espaço.`if``else``elseif``else`

```
if (condition) { // no spaces inside parentheses, space before brace  
    DoOneThing(); // two space indent  
    DoAnotherThing();  
} else if (int a = f(); a != 3) { // closing brace on new line, else on same line  
    DoAThirdThing(a);  
} else {  
    DoNothing();  
}
```

Por razões históricas, permitimos uma exceção às regras acima: se uma declaração não tem ou cláusulas, então as chaves para a declaração controlada ou as quebras de linha dentro das chaves podem ser omitidas se, como resultado, toda a declaração aparecer em uma única linha (nesse caso há um espaço entre o parêntese de fechamento e a declaração controlada) ou em duas linhas (nesse caso há uma quebra de linha após o parêntese de fechamento e não há aparelhos). Por exemplo, os seguintes formulários são permitidos sob esta exceção.`if``else``elseif` `if``if`

```
if (x == kFoo) return new Foo();  
  
if (x == kBar)  
    return new Bar(arg1, arg2, arg3);  
  
if (x == kQuz) { return new Quz(1, 2, 3); }
```

Use este estilo somente quando a declaração for breve, e considere que declarações condicionais com condições complexas ou declarações controladas podem ser mais legíveis com chaves. Alguns projetos requerem chaves sempre.

Finalmente, estes não são permitidos:

```
// Bad - IF statement with ELSE is missing braces  
if (x) DoThis();  
else DoThat();  
  
// Bad - IF statement with ELSE does not have braces everywhere  
if (condition)  
    foo;  
else {  
    bar;  
}  
  
// Bad - IF statement is too long to omit braces  
if (condition)  
    // Comment  
    DoSomething();  
  
// Bad - IF statement is too long to omit braces  
if (condition1 &&  
    condition2)  
    DoSomething();
```

⇒ Loops e Declarações de Switch

As instruções do Switch podem usar aparelhos para blocos. Anotar quedas não triviais entre os casos. Os aparelhos são opcionais para loops de declaração única. Corpos de loop vazios devem usar aparelhos vazios ou `.continue`

case blocos em declarações podem ter chaves ou não, dependendo da sua preferência. Se você incluir chaves, elas devem ser colocadas como mostrado abaixo.switch

Se não estiver condicionado a um valor enumerado, as instruções de switch devem sempre ter um caso (no caso de um valor enumerado, o compilador irá avisá-lo se algum valor não for manuseado). Se o caso padrão nunca for executado, trate-o como um erro. Por exemplo: default

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

O recuo de um rótulo de caso para outro deve ser anotado usando a macro (definida em). deve ser colocado em um ponto de execução onde ocorre uma queda para o próximo rótulo de caso. Uma exceção comum são rótulos de casos consecutivos sem código de intervenção, nesse caso não é necessária anotação.ABSL_FALLTHROUGH_INTENDED;abs1/base/macros.hABSL_FALLTHROUGH_INTENDED;

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

Os aparelhos são opcionais para loops de declaração única.

```
for (int i = 0; i < kSomeNumber; ++i)
    printf("I love you\n");

for (int i = 0; i < kSomeNumber; ++i) {
    printf("I take it back\n");
}
```

Corpos vazios devem usar um par vazio de aparelhos ou sem aparelho, em vez de um único ponto e vírgula.continue

```
while (condition) {
    // Repeat test until it returns false.
}
for (int i = 0; i < kSomeNumber; ++i) {} // Good - one newline is also OK.
while (condition) continue; // Good - continue indicates no logic.
```

```
while (condition); // Bad - looks like part of do/while loop.
```

☞ Ponteiros e Expressões de Referência

Sem espaços ao redor do período ou flecha. Os operadores de ponteiros não têm espaços de arrasto.

A seguir, exemplos de expressões de ponteiro e referência corretamente formatadas:

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

Note que:

- Não há espaços ao redor do período ou seta ao acessar um membro.
- Os operadores de ponteiros não têm espaço após o ou .*&

Ao se referir a um ponteiro ou referência (declarações ou definições variáveis, argumentos, tipos de retorno, parâmetros de modelo, etc), você pode colocar o espaço antes ou depois do asterisco/ampersand. No estilo espaço de trilha, o espaço é elidido em alguns casos (parâmetros de modelo, etc).

```
// 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 '>'
```

Você deve fazer isso consistentemente dentro de um único arquivo. Ao modificar um arquivo existente, use o estilo nesse arquivo.

É permitido (se incomum) declarar múltiplas variáveis na mesma declaração, mas é proibido se algum deles tiver ponteiros ou decorações de referência. Tais declarações são facilmente mal interpretadas.

```
// Fine if helpful for readability.  
int x, y;
```

```
int x, *y; // Disallowed - no & or * in multiple declaration  
int* x, *y; // Disallowed - no & or * in multiple declaration; inconsistent spacing  
char * c; // Bad - spaces on both sides of *  
const std::string & str; // Bad - spaces on both sides of &
```

⇒ Expressões Booleanas

Quando você tem uma expressão booleana que é mais longa do que o [comprimento da linha padrão](#), seja consistente em como você quebra as linhas.

Neste exemplo, o operador e lógico está sempre no final das linhas:

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

Observe que quando o código se envolve neste exemplo, ambos os operadores lógicos estão no final da linha. Isso é mais comum no código do Google, embora embrulhar todos os operadores no início da linha também seja permitido. Sinta-se livre para inserir parênteses extras criteriosamente porque eles podem ser muito úteis para aumentar a legibilidade quando usados adequadamente, mas tenha cuidado com o uso excessivo. Observe também que você deve sempre usar os operadores de pontuação, como e , em vez da palavra operadores, como e .&&&&~andcompl

⇒ Valores de retorno

Não cerque desnecessariamente a expressão com parênteses.`return`

Use parênteses apenas onde você os usaria `.return expr;x = expr;`

```
return result; // No parentheses in the simple case.  
// Parentheses OK to make a complex expression more readable.  
return (some_long_condition &&  
       another_condition);
```

```
return (value); // You wouldn't write var = (value);  
return(result); // return is not a function!
```

⇒ Inicialização variável e array

Você pode escolher entre, e; a seguir estão todos corretos:`=(){}=()`

```
int x = 3;  
int x(3);  
int x{3};
```

```
std::string name = "Some Name";
std::string name("Some Name");
std::string name{"Some Name"};
```

Tenha cuidado ao usar uma lista de inicialização em um tipo com um construtor. Uma *lista de init-init-init não-adocito* prefere o construtor sempre que possível. Observe que os aparelhos vazios são especiais e chamarão um construtor padrão se disponível. Para forçar o não-construtor, use parênteses em vez de aparelhos.`{...}`std::initializer_liststd::initializer_list{std::initializer_list

```
std::vector<int> v(100, 1); // A vector containing 100 items: All 1s.
std::vector<int> v{100, 1}; // A vector containing 2 items: 100 and 1.
```

Além disso, a forma de cinta evita o estreitamento de tipos integrais. Isso pode evitar alguns tipos de erros de programação.

```
int pi(3.14); // OK -- pi == 3.
int pi{3.14}; // Compile error: narrowing conversion.
```

☞ Diretrizes do pré-processador

A marca hash que inicia uma diretiva pré-processador deve estar sempre no início da linha.

Mesmo quando as diretivas do pré-processador estão dentro do corpo do código recuado, as diretivas devem começar no início da linha.

```
// Good - directives at beginning of line
if (lopsided_score) {
#ifndef DISASTER_PENDING // Correct -- Starts at beginning of line
    DropEverything();
#endif NOTIFY // OK but not required -- Spaces after #
    NotifyClient();
#endif
#endif
    BackToNormal();
}
```

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

☞ Formato de classe

Seções em , e ordem, cada um recuado um espaço.publicprotectedprivate

O formato básico para uma definição de classe (sem os comentários, ver [Comentários de Classe](#) para uma discussão sobre quais comentários são necessários) é:

```
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_;
};
```

Coisas a notar:

- Qualquer nome de classe base deve estar na mesma linha do nome da subclasse, sujeito ao limite de 80 colunas.
- As palavras-chave devem ser recuadas em um espaço.`public:protected:private`:
- Exceto na primeira instância, essas palavras-chave devem ser precedidas por uma linha em branco. Essa regra é opcional em classes pequenas.
- Não deixe uma linha em branco após essas palavras-chave.
- A seção deve ser a primeira, seguida pelo e finalmente a seção.`publicprotectedprivate`
- Consulte [a Ordem de Declaração](#) para obter regras sobre a ordenação de declarações em cada uma dessas seções.

⇨ Listas de iniciadores de construtores

As listas de inicializadores de construtor podem ser todas em uma linha ou com linhas subsequentes recuadas quatro espaços.

Os formatos aceitáveis para listas de inicializadores são:

```
// When everything fits on one line:
MyClass::MyClass(int var) : some_var_(var) {
    DoSomething();
}

// If the signature and initializer list are not all on one line,
// you must wrap before the colon and indent 4 spaces:
MyClass::MyClass(int var)
    : some_var_(var), some_other_var_(var + 1) {
    DoSomething();
}

// When the list spans multiple lines, put each member on its own line
// and align them:
MyClass::MyClass(int var)
    : some_var_(var),           // 4 space indent
    some_other_var_(var + 1) {  // lined up
    DoSomething();
}

// As with any other code block, the close curly can be on the same
// line as the open curly, if it fits.
MyClass::MyClass(int var)
    : some_var_(var) {}
```

⇨ Formatação do Namespace

O conteúdo dos namespaces não é recuado.

[Os namespaces](#) não adicionam um nível extra de recuo. Por exemplo, use:

```
namespace {

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

} // namespace
```

Não faça recuo dentro de um namespace:

```
namespace {

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

} // namespace
```

⇨ Espaço branco horizontal

O uso do espaço branco horizontal depende da localização. Nunca coloque o espaço branco no final de uma linha.

Geral

```
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.
    ...
```

Adicionar espaço branco trailing pode causar trabalho extra para outros que editam o mesmo arquivo, quando eles se fundem, assim como remover o espaço branco de arrasto existente. Então: não introduza o espaço branco. Remova-o se você já estiver mudando essa linha ou fazê-la em uma operação de limpeza separada (de preferência quando ninguém mais estiver trabalhando no arquivo).

Laços e Condicionais

```
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) {
// Loops and conditions may have spaces inside parentheses, but this
// is rare. Be consistent.
switch ( i ) {
if ( test ) {
for ( int i = 0; i < 5; ++i ) {
// For loops always have a space after the semicolon. They may have a space
// before the semicolon, but this is rare.
for ( ; i < 5 ; ++i) {
    ...

// Range-based for loops always have a space before and after the colon.
for (auto x : counts) {
    ...
}
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.
```

Operadores

```
// Assignment operators always have spaces around them.
x = 0;

// Other binary operators usually have spaces around them, but it's
// OK to remove spaces around factors. Parentheses should have no
// internal padding.
v = w * x + y / z;
v = w*x + y/z;
v = w * (x + z);

// No spaces separating unary operators and their arguments.
x = -5;
++x;
if (x && !y)
    ...
```

Modelos e Moldes

```
// No spaces inside the angle brackets (< and >), before  
// <, or between >( in a cast  
std::vector<std::string> x;  
y = static_cast<char*>(x);  
  
// Spaces between type and pointer are OK, but be consistent.  
std::vector<char *> x;
```

⇒ Espaço branco vertical

Minimizar o uso de espaço branco vertical.

Isso é mais um princípio do que uma regra: não use linhas em branco quando você não precisa. Em particular, não coloque mais de uma ou duas linhas em branco entre as funções, resista a funções iniciais com uma linha em branco, não termine funções com uma linha em branco e poupe com o uso de linhas em branco. Uma linha em branco dentro de um bloco de código serve como uma quebra de parágrafo em prosa: separar visualmente dois pensamentos.

O princípio básico é: Quanto mais código se encaixa em uma tela, mais fácil é seguir e entender o fluxo de controle do programa. Use o espaço branco propositalmente para fornecer separação nesse fluxo.

Algumas regras para ajudar quando linhas em branco podem ser úteis:

- Linhas em branco no início ou no final de uma função não ajudam na legibilidade.
- Linhas em branco dentro de uma cadeia de blocos se em outra podem muito bem ajudar a legibilidade.
- Uma linha em branco antes de uma linha de comentários geralmente ajuda a legibilidade — a introdução de um novo comentário sugere o início de um novo pensamento, e a linha em branco deixa claro que o comentário acompanha a seguinte coisa em vez do anterior.
- Linhas em brancos imediatamente dentro de uma declaração de um namespace ou bloco de namespaces podem ajudar a legibilidade, separando visualmente o conteúdo de suporte de carga do invólucro organizacional (em grande parte não semântico). Especialmente quando a primeira declaração dentro do namespace(s) é precedida por um comentário, este se torna um caso especial da regra anterior, ajudando o comentário a "anexar" à declaração subsequente.

⇒ Exceções às Regras

As convenções de codificação descritas acima são obrigatórias. No entanto, como todas as boas regras, estas às vezes têm exceções, que discutimos aqui.

⇒ Código não conforme existente

Você pode divergir das regras ao lidar com códigos que não estão em conformidade com este guia de estilo.

Se você se encontrar modificando o código que foi escrito para especificações diferentes das apresentadas por este guia, você pode ter que divergir dessas regras para se manter consistente com as convenções locais nesse código. Se você está em dúvida sobre como fazer isso, pergunte ao autor original ou à pessoa atualmente responsável pelo código. Lembre-se que a *consistência* inclui consistência local, também.

⇒ Código do Windows

Os programadores do Windows desenvolveram seu próprio conjunto de convenções de codificação, principalmente derivadas das convenções em cabeçalhos do Windows e outros códigos microsoft. Queremos facilitar para qualquer pessoa entender seu código, por isso temos um único conjunto de diretrizes para todos que escrevem C++ em qualquer plataforma.

Vale a pena reiterar algumas das diretrizes que você pode esquecer se estiver acostumado com o estilo predominante do Windows:

- Não use notação húngara (por exemplo, nomeando um inteiro). Use as convenções de nomeação do Google, incluindo a extensão para arquivos de origem,.iNum.cc
- O Windows define muitos dos seus próprios sinônimos para tipos primitivos, como, etc. É perfeitamente aceitável e encorajado que você use esses tipos ao chamar as funções de API do Windows. Mesmo assim, mantenha o mais próximo possível dos tipos C+++subjacentes. Por exemplo, use em vez de .DWORDHANDLEconst TCHAR *LPCTSTR

- Ao compilar com o Microsoft Visual C++, defina o compilador para o nível de aviso 3 ou superior e trate todos os avisos como erros.
- Não utilize; em vez disso, use o Google padrão incluir guardas. O caminho na inclusão de guardas deve ser relativo ao topo da árvore do seu projeto.`#pragma once`
- Na verdade, não use nenhuma extensão fora do padrão, como `e`, a menos que você absolutamente deve. O uso `e` é permitido; no entanto, você deve usá-los através de macros como `e`, para que alguém possa facilmente desativar as extensões se eles compartilham o código.`#pragma __declspec __declspec(dllexport) __declspec(dllexport) DLLIMPORTDLLEXPORT`

No entanto, existem apenas algumas regras que ocasionalmente precisamos quebrar no Windows:

- Normalmente, [desencorajamos fortemente o uso de herança de implementação múltipla](#); no entanto, é necessário ao usar com algumas classes ATL/WTL. Você pode usar várias heranças de implementação para implementar classes e interfaces COM ou ATL/WTL.
- Embora você não deva usar exceções em seu próprio código, elas são usadas extensivamente no ATL e em alguns STLs, incluindo a que vem com Visual C++. Ao usar o ATL, você deve definir para desativar exceções. Você deve investigar se você também pode desativar exceções em seu STL, mas se não, não há problema em ativar exceções no compilador. (Note que isso é apenas para obter o STL para compilar. Você ainda não deve escrever o código de manuseio de exceções você mesmo.)`_ATL_NO_EXCEPTIONS`
- A maneira usual de trabalhar com cabeçalhos pré-compensados é incluir um arquivo de cabeçalho na parte superior de cada arquivo de origem, normalmente com um nome como `ou.h`. Para tornar seu código mais fácil de compartilhar com outros projetos, evite incluir este arquivo explicitamente (exceto `in`), e use a opção de compilador para incluir o arquivo automaticamente.`StdAfx.hprecompile.hprecompile.cc/FI`
- Os cabeçalhos de recursos, que geralmente são nomeados e contêm apenas macros, não precisam estar de acordo com essas diretrizes de estilo.`resource.h`