

Contents

[Documentação da linguagem C++](#)

[Referência da linguagem C++](#)

[Referência da linguagem C++](#)

[Bem-vindo de volta ao C++ \(C++ moderno\)](#)

[Convenções lexicais](#)

[Convenções lexicais](#)

[Tokens e conjuntos de caracteres](#)

[Comentários](#)

[Identificadores](#)

[Palavras-chave](#)

[Pontuadores](#)

[Literais de ponteiro, numéricos e boolianos](#)

[Literais de cadeia de caracteres e de caracteres](#)

[Literais definidos pelo usuário](#)

[Conceitos básicos](#)

[Conceitos básicos](#)

[Sistema do tipo C++](#)

[Escopo](#)

[Arquivos de cabeçalho](#)

[Unidades de tradução e vinculação](#)

[função main e argumentos de linha de comando](#)

[Término do programa](#)

[Lvalues e rvalues](#)

[Objetos temporários](#)

[Alinhamento](#)

[Tipos triviais, de layout padrão e de POD](#)

[Tipos de valor](#)

[Conversões e segurança de tipo](#)

[Conversões padrão](#)

Tipos internos

Tipos internos

Intervalos de tipos de dados

nullptr

void

bool

False

True

char, wchar_t, char16_t, char32_t

_int8, _int16, _int32, _int64

_m64

_m128

_m128d

_m128i

_ptr32, _ptr64

Limites numéricos

Limites numéricos

Limites de inteiro

Limites flutuantes

Declarações e definições

Declarações e definições

Classes de armazenamento

auto

const

constexpr

extern

Inicializadores

Aliases e typedefs

usando declaração

volatile

decltype

Atributos

Operadores internos, precedência e associatividade

Operadores internos, precedência e associatividade

operador alignof

Operador _uuidof

Operadores aditivos: + e -

Operador address-of: &

Operadores de atribuição

Operador bitwise AND: &

Operador OR exclusivo bit a bit: ^

Operador OR inclusivo bit a bit: |

Operador cast: ()

Operador de vírgula: ,

Operador condicional: ?:

Operador delete

Operadores de igualdade: == e !=

Operador de conversão de tipo explícito: ()

Operador de chamada da função: ()

Operador de indireção: *

Operadores de deslocamento à esquerda e deslocamento à direita (>> e <<)

Operador AND lógico: &&

Operador de negação lógico: !

Operador OR lógico: ||

Operadores de acesso a membros: . e ->

Operadores multiplicativos e o operador de módulo

Operador new

Operador de complemento individual: ~

Operadores de ponteiro para membro: .* e ->*

Operadores de incremento e de decremento pós-fixados: ++ e --

Operadores de incremento e de decremento pré-fixados: ++ e --

Operadores relacionais: <, >, <=, e >=

Operador de resolução do escopo: ::

Operador sizeof

Operador subscript:

Operador typeid

Operadores unários de adição e de negação: + e -

Expressões

Expressões

Tipos de expressões

Tipos de expressões

Expressões primárias

Reticências e modelos variádicos

Expressões pós-fixadas

Expressões com operadores unários

Expressões com operadores binários

Expressões constantes

Semântica de expressões

Conversão

Conversão

Operadores de conversão

Operadores de conversão

Operador dynamic_cast

Exceção bad_cast

Operador static_cast

Operador const_cast

Operador reinterpret_cast

Informações de tipo de tempo de execução (RTTI)

Informações de tipo de tempo de execução (RTTI)

Exceção bad_typeid

Classe type_info

Instruções

Instruções

Visão geral de instruções C++

Instruções rotuladas

Instrução de expressão

[Instrução de expressão](#)

[Instrução nula](#)

[Instruções compostas \(blocos\)](#)

[Instruções de seleção](#)

[Instruções de seleção](#)

[Instrução if-else](#)

[Instrução __if_exists](#)

[Instrução __if_not_exists](#)

[Instrução switch](#)

[Instruções de iteração](#)

[Instruções de iteração](#)

[Instrução while](#)

[Instrução do-while](#)

[Instrução for](#)

[Instrução for baseada em intervalo](#)

[Instruções de atalho](#)

[Instruções de atalho](#)

[Instrução break](#)

[Instrução continue](#)

[Instrução return](#)

[Instrução goto](#)

[Transferências de controle](#)

[Namespaces](#)

[Enumerações](#)

[Uniões](#)

[Funções](#)

[Funções](#)

[Funções com listas de argumentos variáveis](#)

[Sobrecarga de função](#)

[Funções explicitamente usadas como padrão e excluídas](#)

[Pesquisa de nome dependente do argumento \(Koenig\) em funções](#)

[Argumentos padrão](#)

Funções embutidas

Sobrecarga de operador

Sobrecarga de operador

Regras gerais para sobrecarga de operador

Operadores unários de sobrecarga

Operadores unários de sobrecarga

Sobrecarga de operador de incremento e decremento

Operadores binários

Atribuição

Chamada de função

Subscrito

Acesso de membros

Classes e structs

Classes e structs

classe

struct

Visão geral de membros de classe

Controle de acesso a membro

Controle de acesso a membro

friend

particulares

protegidos

públicos

Inicialização de chaves

Tempo de vida do objeto e gerenciamento de recursos (RAII)

Idioma pimpl para encapsulamento do tempo de compilação

Portabilidade em limites ABI

Construtores

Construtores

Operadores de construtores de cópia e de atribuição de cópia

Operadores de construtores de movimento e de atribuição de movimento

Delegação de construtores

Destruidores

Visão geral das funções de membro

Visão geral das funções de membro

Especificador virtual

Especificador override

Especificador final

Herança

Herança

Funções virtuais

Herança única

Classes base

Várias classes base

Substituições explícitas

Classes abstratas

Resumo das regras de escopo

Palavras-chave de herança

`virtual`

`_super`

`_interface`

Funções de membro especiais

Membros estáticos

Classes C++ como tipos de valor

Conversões de tipo definido pelo usuário

Membros de dados mutáveis

Declarações de classe aninhada

Tipos de classe anônima

Ponteiros para membros

Ponteiro `this`

Campos de bits

Expressões lambda em C++

Expressões lambda em C++

Sintaxe da expressão lambda

[Exemplos de expressões lambda](#)

[Expressões lambda constexpr](#)

[Matrizes](#)

[Referências](#)

[Referências](#)

[Declarador de referência Lvalue: &](#)

[Declarador de referência Rvalue: &&](#)

[Argumentos de funções de tipo de referência](#)

[Retornos de funções de tipo de referência](#)

[Referências a ponteiros](#)

[Ponteiros](#)

[Ponteiros](#)

[Ponteiros brutos](#)

[Ponteiros const e volatile](#)

[Operadores new e delete](#)

[Ponteiros inteligentes](#)

[Como: criar e usar instâncias unique_ptr](#)

[Como: criar e usar instâncias shared_ptr](#)

[Como: criar e usar instâncias weak_ptr](#)

[Como: criar e usar instâncias CComPtr e CComQIPtr](#)

[Pimpl para encapsulamento do tempo de compilação](#)

[Tratamento de exceções em C++](#)

[Tratamento de exceções em C++](#)

[Melhores práticas modernas de C++](#)

[Como projetar tendo em vista a segurança da exceção](#)

[Como realizar a interface entre códigos excepcional e não excepcional](#)

[Instruções try, throw e catch](#)

[Como os blocos catch são avaliados](#)

[Desenrolamento de exceções e de pilha](#)

[Especificações de exceção \(lançar\)](#)

[noexcept](#)

[Exceções C++ não tratadas](#)

Combinação de exceções C (estruturadas) e C++

Combinação de exceções C (estruturadas) e C++

Usando setjmp-longjmp

Tratar exceções estruturadas no C++

SEH (tratamento de exceções estruturado) (C/C++)

SEH (tratamento de exceções estruturado) (C/C++)

Escrevendo um manipulador de exceção

Escrevendo um manipulador de exceção

Instrução try-except

Escrevendo um filtro de exceção

Acionando exceções de software

Exceções de hardware

Restrições em manipuladores de exceção

Escrevendo um manipulador de término

Escrevendo um manipulador de término

Instrução try-finally

Limpando recursos

Tempo de tratamento de exceções: Um resumo

Restrições em manipuladores de término

Transportando exceções entre threads

Asserção e mensagens fornecidas pelo usuário

Asserção e mensagens fornecidas pelo usuário

static_assert

Módulos

Visão geral dos módulos no C++

module, import e export

Modelos

Modelos

typename

Modelos de classe

Modelos de função

Modelos de função

- Instanciação do modelo de função
- Instanciação explícita
- Especialização explícita de modelos de função
- Ordenação parcial de modelos de função
- Modelos de função de membro
- Especialização de modelo
- Modelos e resolução de nome
 - Modelos e resolução de nome
 - Resolução de nome para tipos dependentes
 - Resolução de nome para nomes declarados localmente
 - Resolução de sobrecarga das chamadas de modelo de função
- Organização do código-fonte (modelos do C++)
- Manipulação de eventos
 - Manipulação de eventos
 - `_event`
 - `_hook`
 - `_raise`
 - `_unhook`
 - Tratamento de eventos em C++ nativo
 - Tratamento de eventos em COM
- Modificadores específicos da Microsoft
 - Modificadores específicos da Microsoft
 - Endereçamento baseado
 - Endereçamento baseado
 - Gramática `_based`
 - Ponteiros com base
 - Convenções de chamada
 - Convenções de chamada
 - Passagem de argumento e convenções de nomenclatura
 - Passagem de argumento e convenções de nomenclatura
 - `_cdecl`
 - `_clrcall`

`_stdcall`

`_fastcall`

`_thiscall`

`_vectorcall`

Exemplo de chamada: Protótipo de função e chamada

Exemplo de chamada: Protótipo de função e chamada

Resultados do exemplo de chamada

Chamadas de função naked

Chamadas de função naked

Regras e restrições para funções naked

Considerações para escrever o código de prólogo/epílogo

Coprocessador de ponto flutuante e convenções de chamada

Convenções de chamada obsoletas

restrita (C++ AMP)

Palavra-chave `tile_static`

`_declspec`

`_declspec`

`align`

`allocate`

`allocator`

`appdomain`

`code_seg (_declspec)`

preterido

`dllexport, dllimport`

`dllexport, dllimport`

Definições e declarações

Definindo funções embutidas do C++ com `dllexport` e `dllimport`

Regras e limitações gerais

Usando o `dllimport` e o `dllexport` nas classes do C++

`jit intrinsic`

`naked`

`noalias`

`noinline`

`noreturn`

`nothrow`

`novtable`

`process`

`propriedade`

`restrict`

`safebuffers`

`selectany`

`spectre`

`thread`

`uuid`

`_restrict`

`_sptr, _uptr`

`_unaligned`

`_w64`

`_func_`

[Suporte para COM do compilador](#)

[Suporte para COM do compilador](#)

[Funções globais COM do compilador](#)

[Funções globais COM do compilador](#)

[_com_raise_error](#)

[ConvertStringToBSTR](#)

[ConvertBSTRToString](#)

[_set_com_error_handler](#)

[Classes de suporte COM do compilador](#)

[Classes de suporte COM do compilador](#)

[Classe _bstr_t](#)

[Classe _bstr_t](#)

[Funções de membro _bstr_t](#)

[Funções de membro _bstr_t](#)

[_bstr_t::Assign](#)

[_bstr_t::Attach](#)
[_bstr_t::_bstr_t](#)
[_bstr_t::copy](#)
[_bstr_t::Detach](#)
[_bstr_t::GetAddress](#)
[_bstr_t::GetBSTR](#)
[_bstr_t::length](#)

Operadores [_bstr_t](#)

[Operadores _bstr_t](#)
[_bstr_t::operator =](#)
[_bstr_t::operator +=, +](#)
[_bstr_t::operator !](#)

[Operadores relacionais _bstr_t](#)
[_bstr_t::wchar_t *, _bstr_t::char*](#)

Classe [_com_error](#)

Classe [_com_error](#)

Funções de membro [_com_error](#)

[Funções de membro _com_error](#)
[_com_error::_com_error](#)
[_com_error::Description](#)
[_com_error::Error](#)
[_com_error::ErrorInfo](#)
[_com_error::ErrorMessage](#)
[_com_error::GUID](#)
[_com_error::HelpContext](#)
[_com_error::HelpFile](#)
[_com_error::HRESULTToWCode](#)
[_com_error::Source](#)
[_com_error::WCode](#)
[_com_error::WCodeToHRESULT](#)

Operadores ([_com_error](#))

Operadores ([_com_error](#))

`_com_error::operator =`
`_com_ptr_t class`
`Classe _com_ptr_t`
`Funções de membro _com_ptr_t`
`Funções de membro _com_ptr_t`
`_com_ptr_t::_com_ptr_t`
`_com_ptr_t::AddRef`
`_com_ptr_t::Attach`
`_com_ptr_t::CreateInstance`
`_com_ptr_t::Detach`
`_com_ptr_t::GetActiveObject`
`_com_ptr_t::GetInterfacePtr`
`_com_ptr_t::QueryInterface`
`_com_ptr_t::Release`

`Operadores (_com_ptr_t)`

`Operadores (_com_ptr_t)`

`_com_ptr_t::operator =`

`Operadores relacionais _com_ptr_t`

`Extratores _com_ptr_t`

`Modelos de função relacional`

`Classe _variant_t`

`Classe _variant_t`

`Funções de membro _variant_t`

`Funções de membro _variant_t`

`_variant_t::_variant_t`

`_variant_t::Attach`

`_variant_t::Clear`

`_variant_t::ChangeType`

`_variant_t::Detach`

`_variant_t::SetString`

`Operadores (_variant_t)`

`Operadores (_variant_t)`

`_variant_t::operator =`
Operadores relacionais `_variant_t`
Extratores `_variant_t`
Extensões da Microsoft
Comportamento não padrão
Limites do compilador
Referência de pré-processador C/C++
Referência da biblioteca C++ padrão

Referência da linguagem C++

02/09/2020 • 5 minutes to read • [Edit Online](#)

Essa referência explica a linguagem de programação C++, conforme implementada no compilador do Microsoft C++. A organização se baseia no *manual de referência do C++ anotado* por Margaret Ellis e Bjarne Stroustrup e no padrão ISO/ISO C++ (c/IEC FDIS 14882). As implementações específicas da Microsoft de recursos da linguagem C++ são incluídas.

Para obter uma visão geral das práticas de programação C++ modernas, consulte [Bem-vindo de volta ao C++](#).

Consulte as tabelas a seguir para localizar rapidamente uma palavra-chave ou um operador:

- [Palavras-chave do C++](#)
- [Operadores do C++](#)

Nesta seção

[Convenções léxicas](#)

Elementos lexicais fundamentais de um programa C++: tokens, comentários, operadores, palavras-chave, pontuadores, literais. Além disso, conversão de arquivo, precedência/associatividade de operadores.

[Conceitos básicos](#)

Escopo, vínculo, inicialização e encerramento do programa, classes de armazenamento e tipos.

Tipos internos Os tipos fundamentais que são criados no compilador C++ e seus intervalos de valor.

[Conversões padrão](#)

Conversões de tipo entre tipos internos. Além disso, conversões aritméticas e conversões entre ponteiro, referência e tipos de ponteiro a membro.

Declarações e definições Declarando e Definindo variáveis, tipos e funções.

[Operadores, precedência e capacidade de associação](#)

Os operadores em C++.

[Expressões](#)

Tipos de expressões, semântica de expressões, tópicos de referência sobre operadores, conversão e operadores de conversão, informações de tipo de tempo de execução.

[Expressões lambda](#)

Uma técnica de programação que define implicitamente uma classe de objeto de função e constrói um objeto de função desse tipo de classe.

[Instruções](#)

Expressão, nulo, composto, seleção, iteração, salto e instruções de declaração.

[Classes e structs](#)

Introdução a classes, estruturas e uniões. Além disso, funções de membro, funções de membro especiais, membros de dados, campos de bits, `this` ponteiros, classes aninhadas.

[Uniões](#)

Tipos definidos pelo usuário em que todos os membros compartilham o mesmo local de memória.

[Classes derivadas](#)

Herança única e múltipla, `virtual` funções, várias classes base, classes **abstratas**, regras de escopo. Além disso, `__super` as `__interface` palavras-chave e.

Controle de acesso de membros

Controlando o acesso a membros de classe: `public`, `private`, e `protected` palavras-chave. Funções e classes amigas.

Sobrecarga

Operadores sobrecarregados, regras para sobrecarga de operador.

Tratamento de exceção

Tratamento de exceções C++, SEH (manipulação de exceção estruturada), palavras-chave usadas na escrita de instruções de tratamento de exceções.

Asserção e mensagens fornecidas pelo usuário

`#error` a diretiva, a `static_assert` palavra-chave, a `assert` macro.

Modelo

Especificações de modelo, modelos de função, modelos de classe, `typename` palavra-chave, modelos vs. macros, modelos e ponteiros inteligentes.

Manipulação de eventos

Eventos de declaração e manipuladores de eventos.

Modificadores específicos da Microsoft

Modificadores específicos do Microsoft C++. Endereçamento de memória, convenções de chamada, `naked` funções, atributos de classe de armazenamento estendido (`__declspec`), `__w64` .

Assembler embutido

Usando linguagem de assembly e C++ em `__asm` blocos.

Supporte a COM do compilador

Uma referência a classes específicas da Microsoft e funções globais usadas para oferecer suporte a tipos COM.

Extensões da Microsoft

Extensões da Microsoft para C++.

Comportamento não padrão

Informações sobre o comportamento não padrão do compilador do Microsoft C++.

Bem-vindo outra vez ao C++

Uma visão geral das práticas de programação C++ modernas para escrever programas seguros, corretos e eficientes.

Seções relacionadas

Extensões de componentes para plataformas de runtime

Material de referência sobre como usar o compilador do Microsoft C++ para o .NET de destino.

Referência de build C/C++

Opções do compilador, opções de vinculador e outras ferramentas de compilação.

Referência de pré-processador do C/C++

Material de referência sobre pragmas, diretivas de pré-processador, macros predefinidas e o pré-processador.

Bibliotecas do Visual C++

Uma lista de links para as páginas de início de referência para as várias bibliotecas do Microsoft C++.

Confira também

[Referência da linguagem C](#)

Bem-vindo de volta ao C++ – C++ moderno

02/09/2020 • 17 minutes to read • [Edit Online](#)

Desde sua criação, o C++ tornou-se uma das linguagens de programação mais amplamente usadas do mundo. os programas bem escritos que a utilizam são rápidos e eficientes. A linguagem é mais flexível do que outras linguagens: ela pode funcionar nos níveis mais altos de abstração e no nível do silício. O C++ fornece bibliotecas padrão altamente otimizadas. Ele permite o acesso a recursos de hardware de baixo nível, para maximizar a velocidade e minimizar os requisitos de memória. Usando o C++, você pode criar uma ampla variedade de aplicativos. Jogos, drivers de dispositivo e software científico de alto desempenho. Programas inseridos. Aplicativos cliente do Windows. Até mesmo bibliotecas e compiladores para outras linguagens de programação são escritos em C++.

Um dos requisitos originais da C++ é a compatibilidade com as versões anteriores da linguagem C. Como resultado, o C++ sempre permitiu a programação C-Style, com ponteiros brutos, matrizes, cadeias de caracteres com terminação nula e outros recursos. Eles podem permitir um ótimo desempenho, mas também podem gerar bugs e complexidade. A evolução do C++ tem recursos enfatizados que reduzem muito a necessidade de usar os idiomas do estilo C. Os antigos recursos de programação de C estão lá quando você precisa deles, mas com código C++ moderno você deve precisar deles menos e menos. O código C++ moderno é mais simples, seguro, mais elegante e ainda tão rápido quanto nunca.

As seções a seguir fornecem uma visão geral dos principais recursos do C++ moderno. A menos que indicado de outra forma, os recursos listados aqui estão disponíveis no C++ 11 e posterior. No compilador do Microsoft C++, você pode definir a `/std` opção do compilador para especificar qual versão do padrão usar para seu projeto.

Recursos e ponteiros inteligentes

Uma das principais classes de bugs na programação em estilo C é o *vazamento de memória*. Os vazamentos são frequentemente causados por uma falha ao chamar a `delete` memória que foi alocada com o `new`. O C++ moderno enfatiza o princípio de que a *aquisição de recursos é a inicialização* (RAII). A ideia é simples. Os recursos (memória de heap, identificadores de arquivo, soquetes e assim por diante) devem *pertencer* a um objeto. Esse objeto cria, ou recebe, o recurso recém alocado em seu construtor e o exclui em seu destruidor. O princípio do RAII garante que todos os recursos sejam retornados corretamente para o sistema operacional quando o objeto proprietário sai do escopo.

Para dar suporte à adoção fácil de princípios RAII, a biblioteca padrão C++ fornece três tipos de ponteiro inteligente: `std::unique_ptr`, `std::shared_ptr` e `std::weak_ptr`. Um ponteiro inteligente manipula a alocação e a exclusão da memória que ela possui. O exemplo a seguir mostra uma classe com um membro de matriz que é alocado no heap na chamada para `make_unique()`. As chamadas para `new` e `delete` são encapsuladas pela `unique_ptr` classe. Quando um `widget` objeto sai do escopo, o destruidor `unique_ptr` será invocado e liberará a memória que foi alocada para a matriz.

```

#include <memory>
class widget
{
private:
    std::unique_ptr<int> data;
public:
    widget(const int size) { data = std::make_unique<int>(size); }
    void do_something() {}
};

void functionUsingWidget() {
    widget w(1000000);    // lifetime automatically tied to enclosing scope
                          // constructs w, including the w.data gadget member
    // ...
    w.do_something();
    // ...
} // automatic destruction and deallocation for w and w.data

```

Sempre que possível, use um ponteiro inteligente ao alocar memória heap. Se você precisar usar os operadores novo e excluir explicitamente, siga o princípio de RAII. Para obter mais informações, consulte [tempo de vida do objeto e gerenciamento de recursos \(RAII\)](#).

std::string e std::string_view

As cadeias de caracteres C-style são outra grande fonte de bugs. Usando o `std::string` e `std::wstring` o, você pode eliminar praticamente todos os erros associados a cadeias de caracteres de estilo C. Você também tem o benefício das funções de membro para pesquisar, acrescentar, aguardar e assim por diante. Ambos são altamente otimizados para velocidade. Ao passar uma cadeia de caracteres para uma função que requer somente acesso somente leitura, no C++ 17, você pode usar `std::string_view` para obter um benefício de desempenho ainda maior.

std::vector e outros contêineres de biblioteca padrão

Todos os contêineres de biblioteca padrão seguem o princípio de RAII. Eles fornecem iteradores para a passagem segura de elementos. E eles são altamente otimizados para desempenho e foram totalmente testados quanto à exatidão. Ao usar esses contêineres, você elimina o potencial de bugs ou ineficiências que podem ser introduzidos em estruturas de dados personalizadas. Em vez de matrizes brutas, use `vector` como um contêiner sequencial em C++.

```

vector<string> apples;
apples.push_back("Granny Smith");

```

Use `map` (não `unordered_map`) como o contêiner associativo padrão. Use `set`, `multimap` e `multiset` para degenerações e vários casos.

```

map<string, string> apple_color;
// ...
apple_color["Granny Smith"] = "Green";

```

Quando a otimização de desempenho for necessária, considere o uso de:

- O `array` tipo ao inserir é importante, por exemplo, como um membro de classe.
- Contêineres associativos não ordenados, como `unordered_map`. Elas têm uma sobrecarga por elemento menor e uma pesquisa em tempo constante, mas podem ser mais difíceis de usar de forma correta e

eficiente.

- Classificado `vector`. Para obter mais informações, consulte [Algoritmos](#).

Não use matrizes de estilo C. Para APIs mais antigas que precisam de acesso direto aos dados, use métodos de acessador, como `f(vec.data(), vec.size())`; em vez disso. Para obter mais informações sobre contêineres, consulte [contêineres de biblioteca padrão C++](#).

Algoritmos de biblioteca padrão

Antes de supor que você precisa escrever um algoritmo personalizado para seu programa, primeiro examine os [algoritmos](#) de biblioteca padrão do C++. A biblioteca padrão contém uma variedade cada vez maior de algoritmos para muitas operações comuns, como pesquisa, classificação, filtragem e randomização. A biblioteca de matemática é extensa. A partir do C++ 17, são fornecidas versões paralelas de muitos algoritmos.

Aqui estão alguns exemplos importantes:

- `for_each`, o algoritmo de passagem padrão (juntamente com loops baseados em intervalos `for`).
- `transform`, para modificação não in-loco de elementos de contêiner
- `find_if`, o algoritmo de pesquisa padrão.
- `sort`, `lower_bound` e os outros algoritmos de classificação e pesquisa padrão.

Para escrever um comparador, use estrito `<` e use *lambda nomeados* quando possível.

```
auto comp = [](const widget& w1, const widget& w2)
    { return w1.weight() < w2.weight(); }

sort( v.begin(), v.end(), comp );

auto i = lower_bound( v.begin(), v.end(), comp );
```

`auto` em vez de nomes de tipo explícitos

O C++ 11 introduziu a `auto` palavra-chave para uso em declarações de variáveis, funções e modelos. `auto` informa ao compilador para deduzir o tipo do objeto para que você não precise digitá-lo explicitamente. `auto` é especialmente útil quando o tipo deduzido é um modelo aninhado:

```
map<int,list<string>>::iterator i = m.begin(); // C-style
auto i = m.begin(); // modern C++
```

Loops baseados em intervalo `for`

A iteração em estilo C em matrizes e contêineres é propenso a indexação de erros e também é entediante de digitar. Para eliminar esses erros e tornar seu código mais legível, use loops baseados em intervalo `for` com contêineres de biblioteca padrão e matrizes brutais. Para obter mais informações, consulte [for instrução baseada em intervalo](#).

```
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> v {1,2,3};

    // C-style
    for(int i = 0; i < v.size(); ++i)
    {
        std::cout << v[i];
    }

    // Modern C++:
    for(auto& num : v)
    {
        std::cout << num;
    }
}
```

constexpr expressões em vez de macros

As macros em C e C++ são tokens que são processados pelo pré-processador antes da compilação. Cada instância de um token de macro é substituída pelo seu valor ou expressão definido antes de o arquivo ser compilado. As macros são comumente usadas na programação em estilo C para definir valores constantes de tempo de compilação. No entanto, as macros são propensas a erros e são difíceis de depurar. No C++ moderno, você deve preferir `constexpr` variáveis para constantes de tempo de compilação:

```
#define SIZE 10 // C-style
constexpr int size = 10; // modern C++
```

Inicialização uniforme

No C++ moderno, você pode usar a inicialização de chaves para qualquer tipo. Essa forma de inicialização é especialmente conveniente ao inicializar matrizes, vetores ou outros contêineres. No exemplo a seguir, `v2` é inicializado com três instâncias do `s`. `v3` é inicializado com três instâncias do `s` que são inicializadas usando chaves. O compilador infere o tipo de cada elemento com base no tipo declarado de `v3`.

```

#include <vector>

struct S
{
    std::string name;
    float num;
    S(std::string s, float f) : name(s), num(f) {}
};

int main()
{
    // C-style initialization
    std::vector<S> v;
    S s1("Norah", 2.7);
    S s2("Frank", 3.5);
    S s3("Jeri", 85.9);

    v.push_back(s1);
    v.push_back(s2);
    v.push_back(s3);

    // Modern C++:
    std::vector<S> v2 {s1, s2, s3};

    // or...
    std::vector<S> v3{ {"Norah", 2.7}, {"Frank", 3.5}, {"Jeri", 85.9} };

}

```

Para obter mais informações, consulte [inicialização de chaves](#).

Mover semântica

O C++ moderno fornece *semântica de movimentação*, o que possibilita eliminar cópias de memória desnecessárias. Em versões anteriores do idioma, as cópias eram inevitáveis em determinadas situações. Uma operação de *movimentação* transfere a propriedade de um recurso de um objeto para o seguinte sem fazer uma cópia. Algumas classes têm recursos como memória de heap, identificadores de arquivos e assim por diante. Ao implementar uma classe de propriedade de recurso, você pode definir um *Construtor de movimentação* e *mover o operador de atribuição* para ele. O compilador escolhe esses membros especiais durante a resolução de sobrecarga em situações em que uma cópia não é necessária. Os tipos de contêiner de biblioteca padrão invocarão o Construtor move em objetos, se houver um definido. Para obter mais informações, consulte [mover construtores e mover operadores de atribuição \(C++\)](#).

Expressões lambda

Na programação em estilo C, uma função pode ser passada para outra função usando um *ponteiro de função*. Ponteiros de função são inconvenientes de manter e entender. A função à qual eles se referem pode ser definida em outro lugar no código-fonte, longe do ponto em que ele é invocado. Além disso, eles não são de tipo seguro. O C++ moderno fornece *objetos de função*, classes que substituem o `operator()` operador, o que permite que eles sejam chamados como uma função. A maneira mais conveniente de criar objetos de função é com [expressões lambda](#) embutidas. O exemplo a seguir mostra como usar uma expressão lambda para passar um objeto de função, que a `for_each` função invocará em cada elemento no vetor:

```

std::vector<int> v {1,2,3,4,5};
int x = 2;
int y = 4;
auto result = find_if(begin(v), end(v), [=](int i) { return i > x && i < y; });

```

A expressão lambda `[=](int i) { return i > x && i < y; }` pode ser lida como "função que usa um único argumento do tipo `int` e retorna um booleano que indica se o argumento é maior que `x` e menor que `y`". Observe que as variáveis `x` e `y` do contexto ao redor podem ser usadas no lambda. O `[=]` especifica que essas variáveis são *capturadas* por valor; em outras palavras, a expressão lambda tem suas próprias cópias desses valores.

Exceções

O C++ moderno enfatiza exceções em vez de códigos de erro como a melhor maneira de relatar e tratar condições de erro. Para obter mais informações, consulte [práticas recomendadas do C++ moderno para exceções e tratamento de erros](#).

`std::atomic`

Use o struct da biblioteca padrão C++ `std::atomic` e os tipos relacionados para mecanismos de comunicação entre threads.

`std::variant` (C++ 17)

As uniões são comumente usadas na programação em estilo C para conservar a memória, permitindo que os membros de diferentes tipos ocupem o mesmo local de memória. No entanto, as uniões não são de tipo seguro e estão sujeitas a erros de programação. O C++ 17 apresenta a `std::variant` classe como uma alternativa mais robusta e segura para as uniões. A `std::visit` função pode ser usada para acessar os membros de um `variant` tipo de maneira segura de tipo.

Consulte também

[Referência da linguagem C++](#)

[Expressões lambda](#)

[Biblioteca padrão do C++](#)

[Tabela de conformidade da linguagem Microsoft C++](#)

Convenções lexicais

25/03/2020 • 2 minutes to read • [Edit Online](#)

Esta seção apresenta os elementos fundamentais de um programa em C++. Você usa esses elementos, chamados de "elementos léxicos" ou "tokens", para construir instruções, definições, declarações etc. que são usadas para construir programas completos. Os elementos léxicos a seguir são abordados nesta seção:

- [Tokens e conjuntos de caracteres](#)
- [Comentários](#)
- [Identificadores](#)
- [Palavras-chave](#)
- [Pontuadores](#)
- [Literais numéricos, Booleanos e de ponteiro](#)
- [Cadeias de caracteres e literais de caracteres](#)
- [Literais definidos pelo usuário](#)

Para obter mais informações sobre C++ como os arquivos de origem são analisados, consulte [fases de tradução](#).

Confira também

[Referência da linguagem C++](#)
[Unidades de tradução e vinculação](#)

Tokens e conjuntos de caracteres

08/01/2020 • 7 minutes to read • [Edit Online](#)

O texto de um C++ programa consiste em tokens e *espaços em branco*. Um token é o menor elemento de um programa em C/C++ que é significativo para o compilador. O C++ analisador reconhece esses tipos de tokens:

- Palavras-chave
- Identificadores
- Literais de ponteiro, numéricos e boolianos
- Literais de cadeia de caracteres e de caracteres
- Literais definidos pelo usuário
- Operadores
- Pontuadores

Os tokens são geralmente separados por um *espaço em branco*, que pode ser um ou mais:

- Em branco
- Tabulações horizontais ou verticais
- Novas linhas
- Feeds de formulário
- Comments

Conjunto de caracteres de origem básico

O C++ padrão especifica um *conjunto de caracteres de origem básico* que pode ser usado em arquivos de origem. Para representar caracteres fora desse conjunto, caracteres adicionais podem ser especificados usando um nome de *caractere universal*. A implementação de MSVC permite caracteres adicionais. O *conjunto de caracteres de origem básico* consiste em 96 caracteres que podem ser usados em arquivos de origem. Esse conjunto inclui o caractere de espaço, a guia horizontal, a guia vertical, o feed de formulário e os caracteres de controle de nova linha e esse conjunto de caracteres gráficos:

a b c d e f g h i j k l m n o p q r s t u v w x y z

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

0 1 2 3 4 5 6 7 8 9

_ { } [] # () < > % : ; . ? * + - / ^ & | ~ ! = , \ " '

Seção específica da Microsoft

MSVC inclui o caractere `$` como um membro do conjunto de caracteres de origem básico. O MSVC também permite que um conjunto adicional de caracteres seja usado em arquivos de origem, com base na codificação do arquivo. Por padrão, o Visual Studio armazena arquivos de origem usando a página de código padrão. Quando os arquivos de origem são salvos usando um código de página específico de localidade ou uma página de código Unicode, o MSVC permite que você use qualquer um dos caracteres de uma dessas páginas em seu código-fonte, com exceção dos códigos de controle não explicitamente permitidos no conjunto de caracteres de origem básico. Por exemplo, você pode colocar caracteres japoneses em comentários, identificadores ou literais de cadeia de caracteres se salvar o arquivo usando uma página de código japonesa. MSVC não permite sequências de caracteres que não podem ser convertidas em caracteres multibyte válidos ou pontos de código Unicode. Dependendo das opções do compilador, nem todos os caracteres permitidos podem aparecer em identificadores.

Para obter mais informações, consulte [Identificadores](#).

Fim da seção específica da Microsoft

Nomes de caracteres universais

Como C++ os programas podem usar muito mais caracteres do que aqueles especificados no conjunto de caracteres de origem básico, você pode especificar esses caracteres de forma portátil usando *nomes de caracteres universais*. Um nome de caractere universal consiste em uma sequência de caracteres que representa um ponto de código Unicode. Elas usam duas formas. Use `\UNNNNNNNN` para representar um ponto de código Unicode no formato U + NNNNNNNN, em que NNNNNNNN é o número de ponto de código hexadecimal de oito dígitos. Use `\uNNNN` de quatro dígitos para representar um ponto de código Unicode do formulário U + 0000NNNN.

Os nomes de caracteres universais podem ser usados em identificadores e em literais de cadeia de caracteres e caracteres. Um nome de caractere universal não pode ser usado para representar um ponto de código substituto no intervalo 0xD800-0xDFFF. Em vez disso, use o ponto de código desejado; o compilador gera automaticamente quaisquer substitutos necessários. Restrições adicionais se aplicam aos nomes de caracteres universais que podem ser usados em identificadores. Para obter mais informações, consulte [identificadores](#) e [cadeias de caracteres e literais de caracteres](#).

Seção específica da Microsoft

O compilador C++ da Microsoft trata um caractere em forma de nome de caractere universal e formulário literal de forma intercambiável. Por exemplo, você pode declarar um identificador usando o formato de nome de caractere universal e usá-lo no formato literal:

```
auto \u30AD = 42; // \u30AD is 'ヰ'  
if (ヰ == 42) return true; // \u30AD and ヰ are the same to the compiler
```

O formato dos caracteres estendidos na área de transferência do Windows é específico para as configurações de localidade do aplicativo. Recortar e colar esses caracteres em seu código de outro aplicativo pode introduzir codificações de caracteres inesperadas. Isso pode resultar na análise de erros sem nenhuma causa visível no código. Recomendamos que você defina a codificação do arquivo de origem para uma página de código Unicode antes de colar os caracteres estendidos. Também recomendamos que você use um IME ou o aplicativo de mapa de caracteres para gerar caracteres estendidos.

Fim da seção específica da Microsoft

Conjuntos de caracteres de execução

Os *conjuntos de caracteres de execução* representam os caracteres e as cadeias que podem aparecer em um programa compilado. Esses conjuntos de caracteres consistem em todos os caracteres permitidos em um arquivo de origem e também os caracteres de controle que representam alerta, backspace, retorno de carro e o caractere nulo. O conjunto de caracteres de execução tem uma representação específica de localidade.

Comentários (C++)

25/03/2020 • 2 minutes to read • [Edit Online](#)

Um comentário é um texto que o compilador ignora, mas que é útil para os programadores. Normalmente, os comentários são usados para fazer anotações no código para fins de referência futura. O compilador os trata como espaço em branco. Você pode usar comentários em testes para tornar determinadas linhas de código inativas; no entanto, `#if` / `#endif` diretivas de pré-processador funcionam melhor para isso porque você pode envolver o código que contém comentários, mas não pode aninhar comentários.

Em C++, um comentário é escrito de uma das seguintes maneiras:

- Usando os caracteres `/*` (barra, asterisco), seguidos por qualquer sequência de caracteres (inclusive novas linhas), seguidos pelos caracteres `*/`. Essa sintaxe é a mesma do ANSI C.
- Usando os caracteres `//` (duas barras), seguidos por qualquer sequência de caracteres. Uma nova linha não precedida imediatamente por uma barra invertida encerra essa forma de comentário. Por isso, ele costuma ser chamado de "comentário de linha única".

Os caracteres de comentário (`/*`, `*/` e `//`) não têm nenhum significado especial em uma constante de caractere, uma literal de cadeia de caracteres ou um comentário. Portanto, os comentários que usam a primeira sintaxe não podem ser aninhados.

Confira também

[Convenções lexicais](#)

Identificadores (C++)

02/09/2020 • 6 minutes to read • [Edit Online](#)

Um identificador é uma sequência de caracteres usados para denotar:

- Nome de objeto ou variável
- Nome de classe, estrutura ou união
- Nome do tipo enumerado
- Membro de uma classe, estrutura, união ou enumeração
- Função ou função de membro de classe
- Nome typedef
- Nome do rótulo
- Nome da macro
- Parâmetro da macro

Os caracteres a seguir são permitidos como qualquer caractere de um identificador:

```
_ a b c d e f g h i j k l m  
n o p q r s t u v w x y z  
A B C D E F G H I J K L M  
N O P Q R S T U V W X Y Z
```

Determinados intervalos de nomes de caracteres universais também são permitidos em um identificador. Um nome de caractere universal em um identificador não pode designar um caractere de controle ou um caractere no conjunto de caracteres de origem básica. Para obter mais informações, consulte [conjuntos de caracteres](#). Esses intervalos de números de ponto de código Unicode são permitidos como nomes de caracteres universais para qualquer caractere em um identificador:

- 00A8, 00AA, 00AD, 00AF, 00B2-00B5, 00B7-00BA, 00BC-00BE, 00C0-00D6, 00D8-00F6, 00F8-00FF, 0100-02FF, 0370-167F, 1681-180D, 180F-1DBF, 1E00-1FFF, 200B-200D, 202A-202E, 203F-2040, 2054, 2060-206F, 2070-20CF, 2100-218F, 2460-24FF, 2776-2793, 2C00-2DFF, 2E80-2FFF, 3004-3007, 3021-302F, 3031-303F, 3040-D7FF, F900-FD3D, FD40-FDCF, FDF0-FE1F, FE30-FE44, FE47-FFFD, 10000-1FFFD, 20000-2FFFD, 30000-3FFFD, 40000-4FFFD, 50000-5FFFD, 60000-6FFFD, 70000-7FFFD, 80000-8FFFD, 90000-9FFFD, A0000-AFFFD, B0000-BFFFD, C0000-CFFFD, D0000-DFFFD, E0000-EFFFD

Os caracteres a seguir são permitidos como qualquer caractere em um identificador, exceto o primeiro:

```
0 1 2 3 4 5 6 7 8 9
```

Esses intervalos de números de ponto de código Unicode também são permitidos como nomes de caracteres universais para qualquer caractere em um identificador, exceto o primeiro:

- 0300-036F, 1DC0-1DFF, 20D0-20FF, FE20-FE2F

Específico da Microsoft

Somente os 2048 primeiros caracteres de identificadores do Microsoft C++ são significativos. Os nomes de tipos

definidos pelo usuário são "decorados" pelo compilador para preservar informações de tipo. O nome resultante, incluindo as informações de tipo, não pode ter mais de 2048 caracteres. (Consulte [nomes decorados](#) para obter mais informações.) Os fatores que podem influenciar o comprimento de um identificador decorado são:

- Se o identificador denota um objeto de tipo definido pelo usuário ou um tipo derivado de um tipo definido pelo usuário.
- Se o identificador denota uma função ou um tipo derivado de uma função.
- O número de argumentos para uma função.

O cifrão `$` é um caractere identificador válido no compilador do Microsoft C++ (MSVC). O MSVC também permite que você use os caracteres reais representados pelos intervalos permitidos de nomes de caracteres universais em identificadores. Para usar esses caracteres, você deve salvar o arquivo usando um código de página de codificação de arquivo que os inclui. Este exemplo mostra como os caracteres estendidos e os nomes de caracteres universais podem ser usados de maneira intercambiável em seu código.

```
// extended_identifier.cpp
// In Visual Studio, use File, Advanced Save Options to set
// the file encoding to Unicode codepage 1200
struct テスト          // Japanese 'test'
{
    void トスト() {}  // Japanese 'toast'
};

int main() {
    テスト \u30D1\u30F3;    // Japanese パン 'bread' in UCN form
    パン.トスト();        // compiler recognizes UCN or literal form
}
```

O intervalo de caracteres permitido em um identificador é menos restritivo ao compilar o código C++/CLI. Identificadores no código compilados usando/CLR devem seguir o [padrão ECMA-335: Common Language Infrastructure \(CLI\)](#).

FINAL específico da Microsoft

O primeiro caractere de um identificador deve ser um caractere alfabético, em maiúsculas ou minúsculas, ou em um sublinhado (`_`). Como os identificadores C++ diferenciam maiúsculas de minúsculas, `fileName` é diferente de `FileName`.

Os identificadores não podem ter exatamente a mesma grafia e caixa (maiúscula ou minúscula) que as palavras-chave. Os identificadores que contêm palavras-chave são aceitos. Por exemplo, `Pint` é um identificador legal, embora ele contenha `int`, que é uma palavra-chave.

O uso de dois caracteres de sublinhado sequenciais (`__`) em um identificador, ou um único sublinhado à esquerda seguido por uma letra maiúscula, é reservado para implementações de C++ em todos os escopos. Você deve evitar o uso de um sublinhado inicial seguido por uma letra minúscula para nomes com o escopo de arquivo por causa de possíveis conflitos com identificadores reservados atuais ou futuros.

Confira também

[Convenções léxicas](#)

Palavras-chave (C++)

02/09/2020 • 6 minutes to read • [Edit Online](#)

Palavras-chave são identificadores reservados pré-definidos que têm significados especiais. Eles não podem ser usados como identificadores em seu programa. As palavras-chave a seguir são reservadas para o Microsoft C++. Os nomes com sublinhados e nomes à esquerda especificados para C++/CX e C++/CLI são extensões da Microsoft.

Palavras-chave padrão do C++

`alignas`
`alignof`
`and`^b
`and_eq`^b
`asm`^{um}
`auto`
`bitand`^b
`bitor`^b
`bool`
`break`
`case`
`catch`
`char`
`char8_t`^c
`char16_t`
`char32_t`
`class`
`compl`^b
`concept`^c
`const`
`const_cast`
`consteval`^c
`constexpr`

`constinit`^c
`continue`
`co_await`^c
`co_return`^c
`co_yield`^c
`decltype`
`default`
`delete`
`do`
`double`
`dynamic_cast`
`else`
`enum`

explicit
export ^c
extern
false
float
for
friend
goto
if
inline

int
long
mutable
namespace
new
noexcept
not ^b
not_eq ^b
nullptr
operator
or ^b
or_eq ^b
private
protected
public
register reinterpret_cast
requires ^c
return
short
signed
sizeof
static
static_assert

static_cast
struct
switch
template
this
thread_local
throw
true
try
typedef
typeid
typename
union
unsigned
using mesma

a palavra-chave `asm` específica da Microsoft substitui a sintaxe do `__asm`. `asm` é reservado para compatibilidade com outras implementações de C++, mas não é implementado. Use `__asm` para assembly embutido em destinos x86. O Microsoft C++ não dá suporte ao assembly embutido para outros destinos.

^b os sinônimos do operador estendido são palavras-chave quando `/permissive-` ou `/za` (desabilita as extensões de idioma) são especificadas. Elas não são palavras-chave quando as extensões da Microsoft estão habilitadas.

^c com suporte quando `/std:c++latest` é especificado.

Palavras-chave C++ específicas da Microsoft

Em C++, os identificadores que contêm dois sublinhados consecutivos são reservados para implementações de compilador. A Convenção da Microsoft é preceder palavras-chave específicas da Microsoft com sublinhados duplos. Essas palavras não podem ser usadas como nomes de identificador.

As extensões da Microsoft são ativadas por padrão. Para garantir que seus programas sejam totalmente portáteis, você pode desabilitar extensões da Microsoft especificando a `/permissive-` opção ou `/za` (desabilitar extensões de linguagem) durante a compilação. Essas opções desabilitam algumas palavras-chave específicas da Microsoft.

Quando as extensões do Microsoft são ativadas, você pode usar as palavras-chave específicas da Microsoft em seus programas. Para estar em conformidade com o ANSI, essas palavras-chave são precedidas por um sublinhado duplo. Para compatibilidade com versões anteriores, há suporte para versões de sublinhado simples de muitas das palavras-chave com sublinhado duplo. A `__cdecl` palavra-chave está disponível sem sublinhado à esquerda.

A `__asm` palavra-chave substitui a sintaxe do C++ `asm`. `asm` é reservado para compatibilidade com outras implementações de C++, mas não é implementado. Use `__asm`.

A `__based` palavra-chave tem usos limitados para compilações de destino de 32 bits e 64 bits.

`__alignof` e
`__asm` e
`__assume` e
`__based` e
`__cdecl` e
`__declspec` e
`__event`
`__except` e
`__fastcall` e
`__finally` e
`__forceinline` e

`__hook` d
`__if_exists`
`__if_not_exists`
`__inline` e
`__int16` e
`__int32` e
`__int64` e
`__int8` e

```

__interface
__leave e
__m128

__m128d
__m128i
__m64
__multiple_inheritance e
__ptr32 e
__ptr64 Oriental
__raise
__restrict e
__single_inheritance Oriental
__sptr Oriental
__stdcall e

__super
__thiscall
__unaligned e
__unhook d
__uptr e
__uuidof e
__vectorcall e
__virtual_inheritance e
__w64 e
__wchar_t

```

função intrínseca ^d usada na manipulação de eventos.

^e para compatibilidade com versões anteriores, essas palavras-chave estão disponíveis com dois sublinhados à esquerda e um único sublinhado à esquerda quando as extensões da Microsoft estão habilitadas (o padrão).

Palavras-chave da Microsoft em `_declspec` modificadores

Esses identificadores são atributos estendidos para o `_declspec` modificador. Elas são consideradas palavras-chave dentro desse contexto.

```

align
allocate
allocator
appdomain
code_seg
deprecated

dllexport
dllimport
jitintrinsic
naked
noalias
noinline

noreturn
nothrow

```

novtable

process

property

restrict

safebuffers

selectany

spectre

thread

uuid

Palavras-chave c++/CLI e C++/CX

`__abstract` f

`__box` f

`__delegate` f

`__gc` f

`__identifier`

`__nogc` f

`__noop`

`__pin` f

`__property` f

`__sealed` f

`__try_cast` f

`__value` f

`abstract` g

`array` g

`as_friend`

`delegate` g

`enum class`

`enum struct`

`event` g

`finally`

`for each in`

`gcnew` g

`generic` g

`initonly`

`interface class` g

`interface struct` g

`interior_ptr` g

`literal` g

`new` g

`property` g

`ref class`

`ref struct`

`safecast`

`sealed` g

`typeid`

`value class` g

`value struct`⁹

^f aplicável somente a Managed Extensions for C++. Essa sintaxe foi preferida. Para obter mais informações, consulte [Extensões de componentes para plataformas de runtime](#).

^f aplicável a C++/CLI.

Confira também

[Convenções lexicais](#)

[Operadores, precedência e Associação internos do C++](#)

Pontuadores (C++)

25/03/2020 • 2 minutes to read • [Edit Online](#)

Os pontuadores em C++ têm um significado sintático e semântico para o compilador, mas não especificam uma operação que gera um valor. Alguns pontuadores, isolados ou combinados, também podem ser operadores C++ ou ser significantes para o pré-processador.

Alguns dos caracteres a seguir são considerados pontuadores:

```
! % ^ & * ( ) - + = { } | ~  
[ ] \ ; ' : " < > ? , . / #
```

Os pontuadores [] , () e {} devem aparecer em pares após a [fase 4](#) da tradução.

Confira também

[Convenções lexicais](#)

Literais de ponteiro, numéricos e boolianos

02/09/2020 • 9 minutes to read • [Edit Online](#)

Um literal é um elemento de programa que representa diretamente um valor. Este artigo aborda literais do tipo inteiro, ponto flutuante, booliano e ponteiro. Para obter informações sobre cadeias de caracteres e literais de caracteres, consulte [cadeias de caracteres e literais de caracteres \(C++\)](#). Você também pode definir seus próprios literais com base em qualquer uma dessas categorias. Para obter mais informações, consulte [literais definidos pelo usuário \(C++\)](#)

. Você pode usar literais em muitos contextos, mas mais comumente para inicializar variáveis nomeadas e para passar argumentos para funções:

```
const int answer = 42;           // integer literal
double d = sin(108.87);        // floating point literal passed to sin function
bool b = true;                 // boolean literal
MyClass* mc = nullptr;          // pointer literal
```

Às vezes, é importante dizer ao compilador como interpretar um literal ou qual tipo específico fornecer a ele. Isso é feito acrescentando prefixos ou sufixos ao literal. Por exemplo, o prefixo `0x` informa ao compilador para interpretar o número que o segue como um valor hexadecimal, por exemplo `0x35`. O `ULL` sufixo informa ao compilador para tratar o valor como um `unsigned long long` tipo, como em `5894345ULL`. Consulte as seções a seguir para obter a lista completa de prefixos e sufixos para cada tipo literal.

Literais inteiros

Os literais inteiros começam com um dígito e não têm partes fracionais ou expoentes. Você pode especificar literais inteiros em formato decimal, octal ou hexadecimal. Você pode especificar tipos assinados ou sem sinal e tipos longos ou curtos.

Quando nenhum prefixo ou sufixo estiver presente, o compilador fornecerá um tipo de valor literal inteiro `int` (32 bits), se o valor couber, caso contrário ele fornecerá o tipo `long long` (64 bits).

Para especificar um literal inteiro decimal, comece a especificação com um dígito diferente de zero. Por exemplo:

```
int i = 157;           // Decimal literal
int j = 0198;          // Not a decimal number; erroneous octal literal
int k = 0365;          // Leading zero specifies octal literal, not decimal
int m = 36'000'000 // digit separators make large values more readable
```

Para especificar um literal inteiro octal, inicie a especificação com 0, seguido por uma sequência de dígitos no intervalo de 0 a 7. Os dígitos 8 e 9 são erros na especificação de um literal octal. Por exemplo:

```
int i = 0377; // Octal literal
int j = 0397; // Error: 9 is not an octal digit
```

Para especificar um literal integral hexadecimal, comece a especificação com `0x` ou `0X` (o caso do "x" não importa), seguido por uma sequência de dígitos no intervalo `0` até `9` e `a` (ou `A`) por meio de `f` (ou `F`). Os dígitos hexadecimais de `a` (ou `A`) a `f` (ou `F`) representam valores no intervalo de 10 a 15. Por exemplo:

```
int i = 0x3fff; // Hexadecimal literal
int j = 0X3FFF; // Equal to i
```

Para especificar um tipo não assinado, use o `u` sufixo ou. Para especificar um tipo longo, use o `l` sufixo ou `L`. Para especificar um tipo integral de 64 bits, use o sufixo `LL` ou `ll`. O sufixo `i64` ainda tem suporte, mas não é recomendável. É específico da Microsoft e não é portável. Por exemplo:

```
unsigned val_1 = 328u; // Unsigned value
long val_2 = 0xFFFFFFFFL; // Long value specified
// as hex literal
unsigned long val_3 = 0776745ul; // Unsigned long value
auto val_4 = 108LL; // signed long long
auto val_4 = 0x800000000000000ULL << 16; // unsigned long long
```

Separadores de dígitos: você pode usar o caractere de aspas simples (apóstrofo) para separar valores de local em números maiores para facilitar a leitura dos seres humanos. Os separadores não têm nenhum efeito na compilação.

```
long long i = 24'847'458'121
```

Literais de ponto flutuante

Os literais de ponto flutuante especificam valores que devem ter uma parte fracionária. Esses valores contêm pontos decimais (`.`) e podem conter expoentes.

Os literais de ponto flutuante têm um *significante* (às vezes chamado de *mantissa*), que especifica o valor do número. Eles têm um *expoente*, que especifica a magnitude do número. Eles têm um sufixo opcional que especifica o tipo do literal. O significante é especificado como uma sequência de dígitos seguida por um ponto, seguido por uma sequência opcional de dígitos que representa a parte fracionária do número. Por exemplo:

```
18.46
38.
```

O expoente, se presente, especifica a magnitude do número como uma potência de 10, como mostrado no seguinte exemplo:

```
18.46e0 // 18.46
18.46e1 // 184.6
```

O expoente pode ser especificado usando `e` ou `E`, que têm o mesmo significado, seguido por um sinal opcional (+ ou -) e uma sequência de dígitos. Se um expoente estiver presente, o ponto decimal à direita é desnecessário em números inteiros como `18E0`.

Os literais de ponto flutuante assumem como padrão o tipo `double`. Usando os sufixos `f` ou `l` ou `F` ou `L` (o sufixo não diferencia maiúsculas de minúsculas), o literal pode ser especificado como `float` ou `long double`.

Embora `long double` e `double` tenham a mesma representação, elas não são do mesmo tipo. Por exemplo, você pode ter funções sobrecarregadas, como

```
void func( double );
```

```
void func( long double );
```

Literais booleanos

Os literais booleanos são `true` e `false`.

Literal de ponteiro (C++ 11)

O C++ apresenta o `nullptr` literal para especificar um ponteiro inicializado com zero. No código portátil, `nullptr` deve ser usado em vez de zero de tipo integral ou macros como `NULL`.

Literais binárias (C++ 14)

Um literal binário pode ser especificado pelo uso do `0B` `0b` prefixo ou, seguido por uma sequência de 1 e 0:

```
auto x = 0B0001101 ; // int
auto y = 0b000001 ; // int
```

Evite usar literais como "constantes mágicas"

Você pode usar literais diretamente em expressões e instruções, embora não seja sempre boa prática de programação:

```
if (num < 100)
    return "Success";
```

No exemplo anterior, uma prática melhor é usar uma constante nomeada que transmita um significado claro, por exemplo "MAXIMUM_ERROR_THRESHOLD". E se o valor de retorno "success" for visto pelos usuários finais, poderá ser melhor usar uma constante de cadeia de caracteres nomeada. Você pode manter constantes de cadeia de caracteres em um único local em um arquivo que pode ser localizado em outros idiomas. O uso de constantes nomeadas ajuda você mesmo e outras pessoas a entender a intenção do código.

Confira também

[Convenções lexicais](#)

[Literais de cadeia de caracteres C++](#)

[Literais definidos pelo usuário do C++](#)

Cadeias de caracteres e literais de caracteres (C++)

02/09/2020 • 33 minutes to read • [Edit Online](#)

O C++ dá suporte a vários tipos de cadeia de caracteres e de caracteres e fornece maneiras de expressar valores literais de cada um desses tipos. No código-fonte, você expressa o conteúdo do seu caractere e os literais de cadeia de caracteres usando um conjunto de caracteres. Os nomes de caracteres universais e os caracteres de escape permitem expressar qualquer cadeia de caracteres usando apenas o conjunto de caracteres de origem básico. Um literal de cadeia de caracteres bruta permite evitar o uso de caracteres de escape e pode ser usado para expressar todos os tipos de literais de cadeia de caracteres. Você também pode criar `std::string` literais sem precisar executar etapas adicionais de construção ou conversão.

```
#include <string>
using namespace std::string_literals; // enables s-suffix for std::string literals

int main()
{
    // Character literals
    auto c0 = 'A'; // char
    auto c1 = u8'A'; // char
    auto c2 = L'A'; // wchar_t
    auto c3 = u'A'; // char16_t
    auto c4 = U'A'; // char32_t

    // Multicharacter literals
    auto m0 = 'abcd'; // int, value 0x61626364

    // String literals
    auto s0 = "hello"; // const char*
    auto s1 = u8"hello"; // const char*, encoded as UTF-8
    auto s2 = L"hello"; // const wchar_t*
    auto s3 = u"hello"; // const char16_t*, encoded as UTF-16
    auto s4 = U"hello"; // const char32_t*, encoded as UTF-32

    // Raw string literals containing unescaped \ and "
    auto R0 = R"("Hello \ world")"; // const char*
    auto R1 = u8R"("Hello \ world")"; // const char*, encoded as UTF-8
    auto R2 = LR"("Hello \ world")"; // const wchar_t*
    auto R3 = uR"("Hello \ world")"; // const char16_t*, encoded as UTF-16
    auto R4 = UR"("Hello \ world")"; // const char32_t*, encoded as UTF-32

    // Combining string literals with standard s-suffix
    auto S0 = "hello"s; // std::string
    auto S1 = u8"hello"s; // std::string
    auto S2 = L"hello"s; // std::wstring
    auto S3 = u"hello"s; // std::u16string
    auto S4 = U"hello"s; // std::u32string

    // Combining raw string literals with standard s-suffix
    auto S5 = R"("Hello \ world")"s; // std::string from a raw const char*
    auto S6 = u8R"("Hello \ world")"s; // std::string from a raw const char*, encoded as UTF-8
    auto S7 = LR"("Hello \ world")"s; // std::wstring from a raw const wchar_t*
    auto S8 = uR"("Hello \ world")"s; // std::u16string from a raw const char16_t*, encoded as UTF-16
    auto S9 = UR"("Hello \ world")"s; // std::u32string from a raw const char32_t*, encoded as UTF-32
}
```

Literais de cadeia de caracteres não podem ter nenhum prefixo, ou „ `u8` `L` `u` e `U` prefixos para denotar caracteres estreitos (byte único ou múltiplos bytes), UTF-8, caracteres largos (UCS-2 ou UTF-16), codificações UTF-16 e UTF-32, respectivamente. Um literal de cadeia de caracteres bruta pode ter `R` „ `u8R` `LR` , `uR` e `UR`

prefixos para os equivalentes de versão bruta dessas codificações. Para criar valores temporários ou estáticos `std::string`, você pode usar literais de cadeia de caracteres ou literais de cadeia de caracteres brutos com um `s` sufixo. Para obter mais informações, consulte a seção [literais de cadeia de caracteres](#) abaixo. Para obter mais informações sobre o conjunto de caracteres de origem básico, os nomes de caracteres universais e o uso de caracteres de páginas de código estendidas em seu código-fonte, consulte [conjuntos de caracteres](#).

Literais de caracteres

Um *literal de caractere* é composto de um caractere constante. Ele é representado pelo caractere entre aspas simples. Há cinco tipos de literais de caracteres:

- Literais de caracteres comuns do tipo `char`, por exemplo `'a'`
- Literais de caracteres UTF-8 do tipo `char` (`char8_t` em C++ 20), por exemplo `u8'a'`
- Literais de caractere largo do tipo `wchar_t`, por exemplo `L'a'`
- Literais de caracteres UTF-16 do tipo `char16_t`, por exemplo `u'a'`
- Literais de caracteres UTF-32 do tipo `char32_t`, por exemplo `u'a'`

O caractere usado para um literal de caractere pode ser qualquer caractere, exceto a barra invertida de caracteres reservados (`\`), aspa simples (`'`) ou nova linha. Os caracteres reservados podem ser especificados usando uma sequência de escape. Os caracteres podem ser especificados usando nomes de caracteres universais, desde que o tipo seja grande o suficiente para conter o caractere.

Codificação

Literais de caracteres são codificados de forma diferente com base em seu prefixo.

- Um literal de caractere sem um prefixo é um literal de caractere comum. O valor de um literal de caractere comum contendo um único caractere, sequência de escape ou nome de caractere universal que pode ser representado no conjunto de caracteres de execução tem um valor igual ao valor numérico de sua codificação no conjunto de caracteres de execução. Um literal de caractere comum que contém mais de um caractere, sequência de escape ou nome de caractere universal é um *literal de multicaractere*. Um literal de multicaractere ou um literal de caractere comum que não pode ser representado no conjunto de caracteres de execução tem `int` o tipo, e seu valor é definido pela implementação. Para MSVC, consulte a seção [específica da Microsoft](#) abaixo.
- Um literal de caractere que começa com o `L` prefixo é um literal de caractere largo. O valor de um literal de caractere largo contendo um único caractere, sequência de escape ou nome de caractere universal tem um valor igual ao valor numérico de sua codificação no conjunto de caracteres de execução largo, a menos que o literal de caractere não tenha nenhuma representação no conjunto de caracteres de execução largo; nesse caso, o valor é definido pela implementação. O valor de um literal de caractere largo contendo vários caracteres, sequências de escape ou nomes de caracteres universais é definido pela implementação. Para MSVC, consulte a seção [específica da Microsoft](#) abaixo.
- Um literal de caractere que começa com o `u8` prefixo é um literal de caractere UTF-8. O valor de um literal de caractere UTF-8 contendo um único caractere, sequência de escape ou nome de caractere universal tem um valor igual ao seu valor de ponto de código ISO 10646 se puder ser representado por uma única unidade de código UTF-8 (correspondente aos controles C0 e ao bloco latino Unicode básico). Se o valor não puder ser representado por uma única unidade de código UTF-8, o programa será mal formado. Um literal de caractere UTF-8 contendo mais de um caractere, sequência de escape ou nome de caractere universal é mal formado.
- Um literal de caractere que começa com o `u` prefixo é um literal de caractere UTF-16. O valor de um literal de caractere UTF-16 que contém um único caractere, sequência de escape ou nome de caractere

universal tem um valor igual ao seu valor de ponto de código ISO 10646 se ele puder ser representado por uma única unidade de código UTF-16 (correspondente ao plano de vários idiomas básico). Se o valor não puder ser representado por uma única unidade de código UTF-16, o programa será mal formado. Um literal de caractere UTF-16 contendo mais de um caractere, sequência de escape ou nome de caractere universal é mal formado.

- Um literal de caractere que começa com o `u` prefixo é um literal de caractere UTF-32. O valor de um literal de caractere UTF-32 contendo um único caractere, sequência de escape ou nome de caractere universal tem um valor igual ao seu valor de ponto de código ISO 10646. Um literal de caractere UTF-32 contendo mais de um caractere, sequência de escape ou nome de caractere universal é mal formado.

Sequências de escape

Há três tipos de sequências de escape: simples, octal e hexadecimal. Sequências de escape podem ser qualquer um dos seguintes valores:

VALOR	SEQUÊNCIA DE ESCAPE
nova linha	<code>\n</code>
barra invertida	<code>\\\</code>
tabulação horizontal	<code>\t</code>
ponto de interrogação	<code>? ou \'?</code>
tabulação vertical	<code>\v</code>
aspas simples	<code>\'</code>
backspace	<code>\b</code>
aspas duplas	<code>\"</code>
retorno de carro	<code>\d</code>
o caractere nulo	<code>\0</code>
avanço de página	<code>\f</code>
octal	<code>\ooo</code>
alerta (Bell)	<code>\a</code>
hexadecimal	<code>\xhhh</code>

Uma sequência de escape octal é uma barra invertida seguida por uma sequência de um a três dígitos octais. Uma sequência de escape octal termina no primeiro caractere que não seja um dígito octal, se encontrado antes do terceiro dígito. O maior valor octal possível é `\377`.

Uma sequência de escape hexadecimal é uma barra invertida seguida pelo caractere `x`, seguida por uma sequência de um ou mais dígitos hexadecimais. Os zeros à esquerda são ignorados. Em um literal de caractere U8 ou comum, o valor hexadecimal mais alto é 0xFF. Em um literal de caractere com prefixado L ou u-prefixado, o valor hexadecimal mais alto é 0xFFFF. Em um literal de caractere largo de U-prefixado, o valor hexadecimal mais alto é 0xFFFFFFFF.

Este código de exemplo mostra alguns exemplos de caracteres de escape usando literais de caractere comuns. A mesma sintaxe de sequência de escape é válida para os outros tipos de literal de caractere.

```
#include <iostream>
using namespace std;

int main() {
    char newline = '\n';
    char tab = '\t';
    char backspace = '\b';
    char backslash = '\\';
    char nullChar = '\0';

    cout << "Newline character: " << newline << "ending" << endl;
    cout << "Tab character: " << tab << "ending" << endl;
    cout << "Backspace character: " << backspace << "ending" << endl;
    cout << "Backslash character: " << backslash << "ending" << endl;
    cout << "Null character: " << nullChar << "ending" << endl;
}
/* Output:
Newline character:
ending
Tab character: ending
Backspace character:ending
Backslash character: \ending
Null character: ending
*/
```

O caractere de barra invertida (`\`) é um caractere de continuação de linha quando é colocado no final de uma linha. Se desejar que um caractere de barra invertida apareça como um literal de caractere, você deverá digitar duas barras invertidas em uma linha (`\\\`). Para obter mais informações sobre o caractere de continuação de linha, consulte [fases de tradução](#).

Específico da Microsoft

Para criar um valor a partir de um literal de multicaractere estreito, o compilador converte o caractere ou a sequência de caracteres entre aspas simples em valores de 8 bits dentro de um inteiro de 32 bits. Vários caracteres no literal preenchem bytes correspondentes conforme necessário de ordem superior para ordem baixa. Em seguida, o compilador converte o inteiro para o tipo de destino seguindo as regras usuais. Por exemplo, para criar um `char` valor, o compilador usa o byte de ordem inferior. Para criar um `wchar_t` `char16_t` valor ou, o compilador usa a palavra de ordem inferior. O compilador avisa que o resultado será truncado se algum bit estiver definido acima do byte ou palavra atribuída.

```
char c0      = 'abcd';      // C4305, C4309, truncates to 'd'
wchar_t w0 = 'abcd';      // C4305, C4309, truncates to '\x6364'
int i0      = 'abcd';      // 0x61626364
```

Uma sequência de escape octal que parece conter mais de três dígitos é tratada como uma sequência octal de 3 dígitos, seguida pelos dígitos subsequentes como caracteres em um literal de multicaractere, que pode fornecer resultados surpreendentes. Por exemplo:

```
char c1 = '\100'; // '@'
char c2 = '\1000'; // C4305, C4309, truncates to '0'
```

As sequências de escape que parecem conter caracteres não octais são avaliadas como uma sequência octal até o último caractere octal, seguidos pelos caracteres restantes como os caracteres subsequentes em um literal de caractere. Aviso C4125 será gerado se o primeiro caractere não octal for um dígito decimal. Por exemplo:

```
char c3 = '\009'; // '9'
char c4 = '\089'; // C4305, C4309, truncates to '9'
char c5 = '\qrs'; // C4129, C4305, C4309, truncates to 's'
```

Uma sequência de escape octal que tem um valor mais alto que `\377` causa o erro C2022: '*valor-em-decimal*': muito grande para o caractere.

Uma sequência de escape que parece ter caracteres hexadecimais e não hexadecimais é avaliada como um literal de multicaractere que contém uma sequência de escape hexadecimal até o último caractere hexadecimal, seguido pelos caracteres não hexadecimais. Uma sequência de escape hexadecimal que não contém dígitos hexadecimais causa o erro do compilador C2153: "literais hexadecimais devem ter pelo menos um dígito hexadecimal".

```
char c6 = '\x0050'; // 'P'
char c7 = '\x0pqr'; // C4305, C4309, truncates to 'r'
```

Se um literal de caractere largo prefixado com `L` contém uma sequência de multicaractere, o valor será obtido do primeiro caractere e o compilador gerará o aviso C4066. Os caracteres subsequentes são ignorados, diferentemente do comportamento do literal multicaractere comum equivalente.

```
wchar_t w1 = L'\100'; // L'@
wchar_t w2 = L'\1000'; // C4066 L'@', 0 ignored
wchar_t w3 = L'\009'; // C4066 L'\0', 9 ignored
wchar_t w4 = L'\089'; // C4066 L'\0', 89 ignored
wchar_t w5 = L'\qrs'; // C4129, C4066 L'q' escape, rs ignored
wchar_t w6 = L'\x0050'; // L'P'
wchar_t w7 = L'\x0pqr'; // C4066 L'\0', pqr ignored
```

A seção específica da Microsoft termina aqui.

Nomes de caracteres universais

Em literais de caracteres e literais de cadeia de caracteres nativos (não brutos), qualquer caractere pode ser representado por um nome de caractere universal. Os nomes de caracteres universais são formados por um prefixo `\u` seguido por um ponto de código Unicode de oito dígitos ou por um prefixo `\u` seguido por um ponto de código Unicode de quatro dígitos. Todos os oito ou quatro dígitos, respectivamente, devem estar presentes para criar um nome de caractere universal bem formado.

```
char u1 = 'A'; // 'A'
char u2 = '\101'; // octal, 'A'
char u3 = '\x41'; // hexadecimal, 'A'
char u4 = '\u0041'; // \u UCN 'A'
char u5 = '\U00000041'; // \U UCN 'A'
```

Pares substitutos

Os nomes de caracteres universais não podem codificar valores no intervalo de ponto de código substituto D800-DFFF. Para pares de substitutos Unicode, especifique o nome do caractere universal usando `\UNNNNNNNN`, em que NNNNNNNN é o ponto de código de oito dígitos para o caractere. O compilador gera um par substituto, se necessário.

No C++ 03, a linguagem permitia apenas um subconjunto de caracteres a ser representado por seus nomes de caracteres universais e permitia alguns nomes de caracteres universais que realmente não representaram nenhum caractere Unicode válido. Esse erro foi corrigido no padrão C++ 11. No C++ 11, os literais de caractere e de cadeia de caracteres e os identificadores podem usar nomes de caracteres universais. Para obter mais informações sobre nomes de caracteres universais, consulte [conjuntos de caracteres](#). Para obter mais informações sobre Unicode, consulte [Unicode](#). Para obter mais informações sobre pares substitutos, consulte

pares substitutos e caracteres suplementares.

Literais de cadeia de caracteres

Uma literal de cadeia de caracteres representa uma sequência de caracteres que, juntos, formam uma cadeia de caracteres terminada em nulo. Os caracteres devem ser incluídos entre aspas duplas. Existem os seguintes tipos de literais de cadeias de caracteres:

Literais de cadeia de caracteres estreitas

Um literal de cadeia de caracteres estreito é uma matriz não prefixada, delimitada por aspas duplas e terminada em nulo do tipo `const char[n]`, em que n é o comprimento da matriz em bytes. Um literal de cadeia de caracteres estreito pode conter qualquer caractere gráfico, exceto aspas duplas (") , barra invertida (\) ou caractere de nova linha. Um literal de cadeia de caracteres estreito também pode conter as sequências de escape listadas acima e os nomes de caracteres universais que se ajustam em um byte.

```
const char *narrow = "abcd";  
  
// represents the string: yes\nno  
const char *escaped = "yes\\\\no";
```

Cadeias de caracteres codificadas em UTF-8

Uma cadeia de caracteres codificada em UTF-8 é uma matriz de tipo `U8`, delimitada por aspas duplas e com terminação de nulo `const char[n]`, em que n é o comprimento da matriz codificada em bytes. Um literal de cadeia de caracteres prefixado `U8` pode conter qualquer caractere gráfico, exceto aspas duplas (") , barra invertida (\) ou caractere de nova linha. Um literal de cadeia de caracteres `U8` também pode conter as sequências de escape listadas acima e qualquer nome de caractere universal.

```
const char* str1 = u8"Hello World";  
const char* str2 = u8"\U0001F607 is 0:-)";
```

Literais de cadeia de caracteres largos

Um literal de cadeia de caracteres largo é uma matriz de constante terminada `wchar_t` em nulo que é prefixada por 'L' e contém qualquer caractere gráfico, exceto aspas duplas (") , barra invertida (\) ou caractere de nova linha. Um literal de cadeia de caracteres largo pode conter as sequências de escape listadas acima e qualquer nome de caractere universal.

```
const wchar_t* wide = L"zyxw";  
const wchar_t* newline = L"hello\\ngoodbye";
```

char16_t e char32_t (C++ 11)

O C++ 11 apresenta os `char16_t` tipos de caracteres portáteis (Unicode de 16 bits) e `char32_t` (unicode de 32 bits):

```
auto s3 = u"hello"; // const char16_t*  
auto s4 = U"hello"; // const char32_t*
```

Literais de cadeia de caracteres brutas (C++ 11)

Um literal de cadeia de caracteres bruta é uma matriz com terminação nula, de qualquer tipo de caractere, que contém qualquer caractere gráfico, incluindo aspas duplas (") , barra invertida (\) ou caractere de nova linha. As literais de cadeias de caracteres brutas costumam ser usadas em expressões regulares que utilizam classes de caracteres, bem como em cadeias de caracteres HTML e XML. Para obter exemplos, consulte o seguinte artigo: [perguntas frequentes sobre o Bjarne Stroustrup no C++ 11](#).

```
// represents the string: An unescaped \ character
const char* raw_narrow = R"(An unescaped \ character)";
const wchar_t* raw_wide = LR"(An unescaped \ character)";
const char* raw_utf8 = u8R"(An unescaped \ character)";
const char16_t* raw_utf16 = uR"(An unescaped \ character)";
const char32_t* raw_utf32 = UR"(An unescaped \ character);
```

Um delimitador é uma sequência definida pelo usuário de até 16 caracteres que precede imediatamente o parêntese de abertura de um literal de cadeia de caracteres bruta e imediatamente segue o parêntese de fechamento. Por exemplo, na `R"abc(Hello"\()abc"` sequência de delimitador é `\()` e o conteúdo da cadeia de caracteres é `Hello"\()`. Você pode usar um delimitador para eliminar a ambiguidade de cadeias de caracteres brutas que contenham aspas duplas e parênteses. Esse literal de cadeia de caracteres causa um erro de compilador:

```
// meant to represent the string: ")"
const char* bad_parens = R"()""; // error C2059
```

Mas um delimitador resolve essa sintaxe:

```
const char* good_parens = R"xyz()"xyz";
```

Você pode construir um literal de cadeia de caracteres bruta que contenha uma nova linha (não o caractere de escape) na fonte:

```
// represents the string: hello
//goodbye
const wchar_t* newline = LR"(hello
goodbye)";
```

literais std:: String (C++ 14)

`std::string` literais são implementações de biblioteca padrão de literais definidos pelo usuário (veja abaixo) que são representadas como `"xyz"s` (com um `s` sufixo). Esse tipo de literal de cadeia de caracteres produz um objeto temporário do tipo `std::string`, `std::wstring`, `std::u32string` ou `std::u16string`, dependendo do prefixo especificado. Quando nenhum prefixo é usado, como acima, um `std::string` é produzido. `L"xyz"s` produz um `std::wstring`. `u"xyz"s` produz um `std::u16string` `U"xyz"s` produz um `std::u32string`.

```
//#include <string>
//using namespace std::string_literals;
string str{ "hello"s };
string str2{ u8"Hello World" };
wstring str3{ L"hello"s };
u16string str4{ u"hello"s };
u32string str5{ U"hello"s };
```

O `s` sufixo também pode ser usado em literais de cadeia de caracteres brutas:

```
u32string str6{ UR"(She said \"hello.\")"s };
```

`std::string` os literais são definidos no namespace `std::literals::string_literals` no `<string>` arquivo de cabeçalho. Como `std::literals::string_literals`, e `std::literals` são declarados como [namespaces embutidos](#), `std::literals::string_literals` é automaticamente tratado como se ele pertencia diretamente no

```
namespace std .
```

Tamanho de literais de cadeia de caracteres

Para cadeias de caracteres ANSI `char*` e outras codificações de byte único (mas não UTF-8), o tamanho (em bytes) de um literal de cadeia de caracteres é o número de caracteres mais 1 para o caractere nulo de terminação. Para todos os outros tipos de cadeia de caracteres, o tamanho não é estritamente relacionado ao número de caracteres. O UTF-8 usa até quatro `char` elementos para codificar algumas *unidades de código*, `char16_t` ou `wchar_t` codificado como UTF-16, pode usar dois elementos (para um total de quatro bytes) para codificar uma única *unidade de código*. Este exemplo mostra o tamanho de um literal de cadeia de caracteres largo em bytes:

```
const wchar_t* str = L"Hello!";
const size_t byteSize = (wcslen(str) + 1) * sizeof(wchar_t);
```

Observe que `strlen()` `wcslen()`, e não incluem o tamanho do caractere nulo de terminação, cujo tamanho é igual ao tamanho do elemento do tipo de cadeia de caracteres: um byte em uma `char*` cadeia de caracteres ou `char8_t*`, dois bytes em `wchar_t*` ou `char16_t*` cadeias, e quatro bytes em `char32_t*` cadeias de caracteres.

O comprimento máximo de um literal de cadeia de caracteres é 65.535 bytes. Esse limite se aplica às literais de cadeias de caracteres estreitas e largas.

Modificando literais de cadeia de caracteres

Como literais de cadeia de caracteres (não incluindo `std::string` literais) são constantes, tentar modificá-las — por exemplo, `str[2] = 'A'` — causa um erro do compilador.

Específico da Microsoft

No Microsoft C++, você pode usar um literal de cadeia de caracteres para inicializar um ponteiro para `const char` ou `wchar_t`. Essa inicialização `const` é permitida no código C99, mas é preterida no C++ 98 e removida no C++ 11. Uma tentativa de modificar a cadeia de caracteres causa uma violação de acesso, como neste exemplo:

```
wchar_t* str = L"hello";
str[2] = L'a'; // run-time error: access violation
```

Você pode fazer com que o compilador emita um erro quando um literal de cadeia de caracteres é convertido em um ponteiro de caractere `const` quando você define a opção de compilador [/zc:strictStrings](#) (desabilitar conversão de tipo literal de cadeia de caracteres). É recomendável para código portável compatível com padrões. Também é uma boa prática usar a `auto` palavra-chave para declarar ponteiros inicializados literais de cadeia de caracteres, pois ele resolve para o tipo correto (`const`). Por exemplo, este exemplo de código captura uma tentativa de gravar em um literal de cadeia de caracteres em tempo de compilação:

```
auto str = L"hello";
str[2] = L'a'; // C3892: you cannot assign to a variable that is const.
```

Em alguns casos, literais de cadeias de caracteres idênticas podem ser agrupadas para economizar espaço no arquivo executável. Em pools de literais de cadeias de caracteres, o compilador faz com que todas as referências a uma literal de cadeia de caracteres específica apontem para o mesmo local na memória, em vez de cada referência apontar para uma instância separada da literal. Para habilitar o pool de cadeias de caracteres, use a [/GF](#) opção do compilador.

A seção [específica da Microsoft](#) termina aqui.

Concatenando literais de cadeias de caracteres adjacentes

Literais de cadeia de caracteres adjacentes ou estreitos são concatenados. Esta declaração:

```
char str[] = "12" "34";
```

é idêntica a esta declaração:

```
char atr[] = "1234";
```

e a esta declaração:

```
char atr[] = "12\  
34";
```

O uso de códigos de escape hexadecimais incorporados para especificar literais de cadeia de caracteres pode causar resultados inesperados. O exemplo a seguir visa criar uma literal de cadeia de caracteres que contenha o caractere ASCII 5, seguido pelos caracteres "f", "i", "v" e "e":

```
"\x05five"
```

O resultado real é um 5F hexadecimal, que é o código ASCII de um sublinhado, seguido pelos caracteres i, v e e. Para obter o resultado correto, você pode usar uma destas sequências de escape:

```
"\005five" // Use octal literal.  
"\x05" "five" // Use string splicing.
```

`std::string` literais, porque são `std::string` tipos, podem ser concatenados com o `+` operador que é definido para `basic_string` tipos. Eles também podem ser concatenados da mesma maneira que os literais de cadeia de caracteres adjacentes. Em ambos os casos, a codificação de cadeia de caracteres e o sufixo devem corresponder:

```
auto x1 = "hello" " " " world"; // OK  
auto x2 = U"hello" " " L"world"; // C2308: disagree on prefix  
auto x3 = u8"hello" " "s u8"world"s; // OK, agree on prefixes and suffixes  
auto x4 = u8"hello" " "s u8"world"z; // C3688, disagree on suffixes
```

Literais de cadeia de caracteres com nomes de caracteres universais

Literais de cadeia de caracteres nativos (não brutos) podem usar nomes de caracteres universais para representar qualquer caractere, desde que o nome do caractere universal possa ser codificado como um ou mais caracteres no tipo de cadeia de caracteres. Por exemplo, um nome de caractere universal representando um caractere estendido não pode ser codificado em uma cadeia de caracteres estreita usando a página de código ANSI, mas pode ser codificado em cadeias estreitas em algumas páginas de código de vários bytes, ou em cadeias de caracteres UTF-8, ou em uma cadeia extensa. No C++ 11, o suporte a Unicode é estendido pelos `char16_t*` `char32_t*` tipos de cadeia de caracteres e:

```
// ASCII smiling face
const char*      s1 = ":-)";

// UTF-16 (on Windows) encoded WINKING FACE (U+1F609)
const wchar_t*   s2 = L"\u0001F609 is ;-)";

// UTF-8 encoded SMILING FACE WITH HALO (U+1F607)
const char*      s3 = u8"\u0001F607 is 0:-)";

// UTF-16 encoded SMILING FACE WITH OPEN MOUTH (U+1F603)
const char16_t*   s4 = u"\u0001F603 is :-D";

// UTF-32 encoded SMILING FACE WITH SUNGLASSES (U+1F60E)
const char32_t*   s5 = U"\u0001F60E is B-)";
```

Confira também

Conjuntos de caracteres

Literais numéricos, Booleanos e de ponteiro

Literais definidos pelo usuário

Literais definidos pelo usuário

02/09/2020 • 9 minutes to read • [Edit Online](#)

Há seis categorias principais de literais em C++: inteiro, caractere, ponto flutuante, Cadeia de caracteres, booliano e ponteiro. A partir do C++ 11, você pode definir seus próprios literais com base nessas categorias, para fornecer atalhos sintáticos para linguagens comuns e aumentar a segurança do tipo. Por exemplo, digamos que você tenha uma `Distance` classe. Você pode definir um literal para quilômetros e outro para milhas e encorajar o usuário a ser explícito sobre as unidades de medida escrevendo: `auto d = 42.0_km` ou `auto d = 42.0_mi`. Não há vantagem de desempenho ou desvantagem de literais definidos pelo usuário; Eles são principalmente para conveniência ou para dedução de tipo em tempo de compilação. A biblioteca padrão tem literais definidos pelo usuário para `std::string`, para `std::complex`, e para as unidades nas operações de tempo e duração no `<chrono>` cabeçalho:

```
Distance d = 36.0_mi + 42.0_km;           // Custom UDL (see below)
std::string str = "hello"s + "World"s; // Standard Library <string> UDL
complex<double> num =
  (2.0 + 3.01i) * (5.0 + 4.3i);        // Standard Library <complex> UDL
auto duration = 15ms + 42h;           // Standard Library <chrono> UDLs
```

Assinaturas de operador literal definidas pelo usuário

Você implementa um literal definido pelo usuário definindo um **operador ""** no escopo de namespace com um dos seguintes formulários:

```
ReturnType operator "" _a(unsigned long long int); // Literal operator for user-defined INTEGRAL literal
ReturnType operator "" _b(long double);           // Literal operator for user-defined FLOATING literal
ReturnType operator "" _c(char);                  // Literal operator for user-defined CHARACTER literal
ReturnType operator "" _d(wchar_t);               // Literal operator for user-defined CHARACTER literal
ReturnType operator "" _e(char16_t);              // Literal operator for user-defined CHARACTER literal
ReturnType operator "" _f(char32_t);              // Literal operator for user-defined CHARACTER literal
ReturnType operator "" _g(const char*, size_t);  // Literal operator for user-defined STRING literal
ReturnType operator "" _h(const wchar_t*, size_t); // Literal operator for user-defined STRING literal
ReturnType operator "" _i(const char16_t*, size_t); // Literal operator for user-defined STRING literal
ReturnType operator "" _g(const char32_t*, size_t); // Literal operator for user-defined STRING literal
ReturnType operator "" _r(const char*);           // Raw literal operator
template<char...> ReturnType operator "" _t();    // Literal operator template
```

Os nomes de operador no exemplo anterior são espaços reservados para qualquer nome que você fornecer; no entanto, o sublinhado à esquerda é necessário. (Somente a biblioteca padrão tem permissão para definir literais sem o sublinhado.) O tipo de retorno é onde você personaliza a conversão ou outras operações feitas pelo literal. Além disso, qualquer um desses operadores pode ser definido como `constexpr`.

Literais cooked

No código-fonte, qualquer literal, seja definido pelo usuário ou não, é essencialmente uma sequência de caracteres alfanuméricos, como `101`, ou ou `54.7` `"hello"` `true`. O compilador interpreta a sequência como um número inteiro, flutuante, Cadeia de * caracteres const char e assim por diante. Um literal definido pelo usuário que aceita como entrada, qualquer tipo que o compilador atribuído ao valor literal é informalmente conhecido como um *literal cooked*. Todos os operadores acima, exceto `_r` e `_t` são literais cooked. Por exemplo, um literal se `42.0_km` associaria a um operador chamado `_km` que tinha uma assinatura semelhante a `_B` e o literal `42_km` seria associado a um operador com uma assinatura semelhante a `_A`.

O exemplo a seguir mostra como os literais definidos pelo usuário podem encorajar os chamadores a serem explícitos sobre sua entrada. Para construir um `Distance`, o usuário deve especificar os quilômetros ou milhas explicitamente usando o literal definido pelo usuário apropriado. Você pode obter o mesmo resultado de outras maneiras, mas os literais definidos pelo usuário são menos detalhados do que as alternativas.

```
// UDL_Distance.cpp

#include <iostream>
#include <string>

struct Distance
{
private:
    explicit Distance(long double val) : kilometers(val)
    {}

    friend Distance operator"" _km(long double val);
    friend Distance operator"" _mi(long double val);

    long double kilometers{ 0 };
public:
    const static long double km_per_mile;
    long double get_kilometers() { return kilometers; }

    Distance operator+(Distance other)
    {
        return Distance(get_kilometers() + other.get_kilometers());
    }
};

const long double Distance::km_per_mile = 1.609344L;

Distance operator"" _km(long double val)
{
    return Distance(val);
}

Distance operator"" _mi(long double val)
{
    return Distance(val * Distance::km_per_mile);
}

int main()
{
    // Must have a decimal point to bind to the operator we defined!
    Distance d{ 402.0_km }; // construct using kilometers
    std::cout << "Kilometers in d: " << d.get_kilometers() << std::endl; // 402

    Distance d2{ 402.0_mi }; // construct using miles
    std::cout << "Kilometers in d2: " << d2.get_kilometers() << std::endl; // 646.956

    // add distances constructed with different units
    Distance d3 = 36.0_mi + 42.0_km;
    std::cout << "d3 value = " << d3.get_kilometers() << std::endl; // 99.9364

    // Distance d4(90.0); // error constructor not accessible

    std::string s;
    std::getline(std::cin, s);
    return 0;
}
```

O número literal deve usar um decimal. Caso contrário, o número seria interpretado como um inteiro e o tipo não seria compatível com o operador. Para entrada de ponto flutuante, o tipo deve ser `long double` e para tipos inteiros, ele deve ser `long long`.

Literais brutos

Em um literal definido pelo usuário bruto, o operador que você define aceita o literal como uma sequência de valores char. Cabe a você interpretar essa sequência como um número ou uma cadeia de caracteres ou outro tipo. Na lista de operadores mostrados anteriormente nesta página, `_r` e `_t` pode ser usado para definir literais brutos:

```
ReturnType operator "" _r(const char*);           // Raw literal operator
template<char...> ReturnType operator "" _t();      // Literal operator template
```

Você pode usar literais brutos para fornecer uma interpretação personalizada de uma sequência de entrada diferente do comportamento normal do compilador. Por exemplo, você pode definir um literal que converte a sequência `4.75987` em um tipo decimal personalizado em vez de um tipo de ponto flutuante de IEEE 754. Literais brutos, como literais `cooked`, também podem ser usados para a validação de tempo de compilação de sequências de entrada.

Exemplo: limitações de literais brutos

O operador literal bruto e o modelo de operador literal só funcionam para literais inteiros e de ponto flutuante definidos pelo usuário, conforme mostrado no exemplo a seguir:

```
#include <cstddef>
#include <cstdio>

// Literal operator for user-defined INTEGRAL literal
void operator "" _dump(unsigned long long int lit)
{
    printf("operator \"\" _dump(unsigned long long int) : ===>%llu<===\n", lit);
};

// Literal operator for user-defined FLOATING literal
void operator "" _dump(long double lit)
{
    printf("operator \"\" _dump(long double) : ===>%Lf<===\n", lit);
};

// Literal operator for user-defined CHARACTER literal
void operator "" _dump(char lit)
{
    printf("operator \"\" _dump(char) : ===>%c<===\n", lit);
};

void operator "" _dump(wchar_t lit)
{
    printf("operator \"\" _dump(wchar_t) : ===>%d<===\n", lit);
};

void operator "" _dump(char16_t lit)
{
    printf("operator \"\" _dump(char16_t) : ===>%d<===\n", lit);
};

void operator "" _dump(char32_t lit)
{
    printf("operator \"\" _dump(char32_t) : ===>%d<===\n", lit);
};

// Literal operator for user-defined STRING literal
void operator "" _dump(const char* lit, size_t)
{
    printf("operator \"\" _dump(const char*, size_t): ===>%s<===\n", lit);
};
```

```

void operator "" _dump(const wchar_t* lit, size_t)
{
    printf("operator \"\" _dump(const wchar_t*, size_t): ===>%ls<===\n", lit);
}

void operator "" _dump(const char16_t* lit, size_t)
{
    printf("operator \"\" _dump(const char16_t*, size_t):\n" );
}

void operator "" _dump(const char32_t* lit, size_t)
{
    printf("operator \"\" _dump(const char32_t*, size_t):\n" );
}

// Raw literal operator
void operator "" _dump_raw(const char* lit)
{
    printf("operator \"\" _dump_raw(const char*) : ===>%s<===\n", lit);
}

template<char...> void operator "" _dump_template(); // Literal operator template

int main(int argc, const char* argv[])
{
    42_dump;
    3.1415926_dump;
    3.14e+25_dump;
    'A'_dump;
    L'B'_dump;
    u'C'_dump;
    U'D'_dump;
    "Hello World"_dump;
    L"Wide String"_dump;
    u8"UTF-8 String"_dump;
    u"UTF-16 String"_dump;
    U"UTF-32 String"_dump;
    42_dump_raw;
    3.1415926_dump_raw;
    3.14e+25_dump_raw;

    // There is no raw literal operator or literal operator template support on these types:
    // 'A'_dump_raw;
    // L'B'_dump_raw;
    // u'C'_dump_raw;
    // U'D'_dump_raw;
    // "Hello World"_dump_raw;
    // L"Wide String"_dump_raw;
    // u8"UTF-8 String"_dump_raw;
    // u"UTF-16 String"_dump_raw;
    // U"UTF-32 String"_dump_raw;
}

```

```
operator "" _dump(unsigned long long int) : ====>42<===
operator "" _dump(long double) : ====>3.141593<===
operator "" _dump(long double) : ====>3139999999999998506827776.000000<===
operator "" _dump(char) : ====>A<===
operator "" _dump(wchar_t) : ====>66<===
operator "" _dump(char16_t) : ====>67<===
operator "" _dump(char32_t) : ====>68<===
operator "" _dump(const char*, size_t): ====>Hello World<===
operator "" _dump(const wchar_t*, size_t): ====>Wide String<===
operator "" _dump(const char*, size_t): ====>UTF-8 String<===
operator "" _dump(const char16_t*, size_t):
operator "" _dump(const char32_t*, size_t):
operator "" _dump_raw(const char*) : ====>42<===
operator "" _dump_raw(const char*) : ====>3.1415926<===
operator "" _dump_raw(const char*) : ====>3.14e+25<===
```

Conceitos básicos (C++)

25/03/2020 • 2 minutes to read • [Edit Online](#)

Esta seção explica os conceitos que são essenciais para entender o C++. Os programadores de linguagem C conhecerão muitos desses conceitos, mas há algumas diferenças sutis que podem causar resultados inesperados de programa. Os tópicos a seguir estão incluídos:

- [C++sistema de tipos](#)
- [Escopo](#)
- [Unidades de tradução e vinculação](#)
- [argumentos da função principal e da linha de comando](#)
- [Encerramento do programa](#)
- [Lvalues e rvalues](#)
- [Objetos temporários](#)
- [Alinhamento](#)
- [Tipos triviais, de layout padrão e POD](#)

Confira também

[Referência da linguagem C++](#)

Sistema do tipo C++

02/09/2020 • 26 minutes to read • [Edit Online](#)

O conceito do *tipo* é muito importante em C++. Cada variável, argumento de função e valor de retorno de função deve ter um tipo para ser compilado. Além disso, cada expressão (incluindo valores literais) recebe implicitamente um tipo do compilador antes de ser avaliada. Alguns exemplos de tipos incluem `int` para armazenar valores inteiros, `double` para armazenar valores de ponto flutuante (também conhecidos como tipos de dados *escalares*) ou a classe de biblioteca padrão `std::basic_string` para armazenar texto. Você pode criar seu próprio tipo definindo um `class` ou `struct`. O tipo especifica a quantidade de memória que será atribuída à variável (ou o resultado da expressão), os tipos de valores que podem ser armazenados nessa variável, como os valores (como padrão de bits) são interpretados e as operações que podem ser executadas nele. Este artigo contém uma visão geral informal dos principais recursos do sistema de tipos C++.

Terminologia

Variável: o nome simbólico de uma quantidade de dados para que o nome possa ser usado para acessar os dados aos quais ele se refere em todo o escopo do código onde ele está definido. Em C++, a *variável* geralmente é usada para fazer referência a instâncias de tipos de dados escalares, enquanto as instâncias de outros tipos geralmente são chamadas de *objetos*.

Objeto: para simplificar e consistência, este artigo usa o termo *objeto* para se referir a qualquer instância de uma classe ou estrutura, e quando ele é usado no sentido geral inclui todos os tipos, até mesmo variáveis escalares.

Tipo de Pod (dados antigos simples): essa categoria informal de tipos de dados em C++ refere-se a tipos que são escalares (consulte a seção tipos fundamentais) ou que são *classes Pod*. Uma classe POD não tem membros de dados estáticos que também não sejam PODs e não tem construtores definidos pelo usuário, destruidores definidos pelo usuário ou operadores de atribuição definidos pelo usuário. Além disso, uma classe POD não tem função virtual, nenhuma classe base e nenhum membro de dados não estático particular ou protegido. Os tipos POD são quase sempre usados para a troca de dados externos, por exemplo, com um módulo escrito na linguagem C (que tem apenas tipos POD).

Especificando tipos de variável e de função

C++ é uma linguagem *fortemente tipada* e também é *digitada estaticamente*; cada objeto tem um tipo e esse tipo nunca é alterado (não deve ser confundido com objetos de dados estáticos). Quando você declara uma variável em seu código, deve especificar seu tipo explicitamente ou usar a `auto` palavra-chave para instruir o compilador a deduzir o tipo do inicializador. Quando você declara uma função em seu código, deve especificar o tipo de cada argumento e seu valor de retorno, ou `void` se nenhum valor for retornado pela função. A exceção se dá quando você está usando modelos de função, que permitem argumentos de tipos arbitrários.

Após a primeira declaração de uma variável, você não pode alterar seu tipo em um momento posterior. No entanto, você pode copiar o valor da variável ou o valor de retorno de uma função para outra variável de tipo diferente. Essas operações são chamadas de *conversões de tipo*, que às vezes são necessárias, mas também são fontes potenciais de perda ou incorreção de dados.

Ao declarar uma variável do tipo POD, é altamente recomendável inicializá-la, ou seja, dar a ela um valor inicial. Até que você inicialize uma variável, ela terá o valor “garbage”, que consiste nos bits que estavam nesse local de memória anteriormente. Esse é um aspecto importante da linguagem C++ a ser lembrado, especialmente se você estiver vindo de outra linguagem que manipule a inicialização para você. Ao declarar uma variável do tipo de classe não POD, o construtor manipula a inicialização.

O exemplo a seguir mostra algumas declarações de variável simples com algumas descrições para cada uma. O exemplo também mostra como o compilador usa informações de tipo para permitir ou não permitir determinadas operações subsequentes na variável.

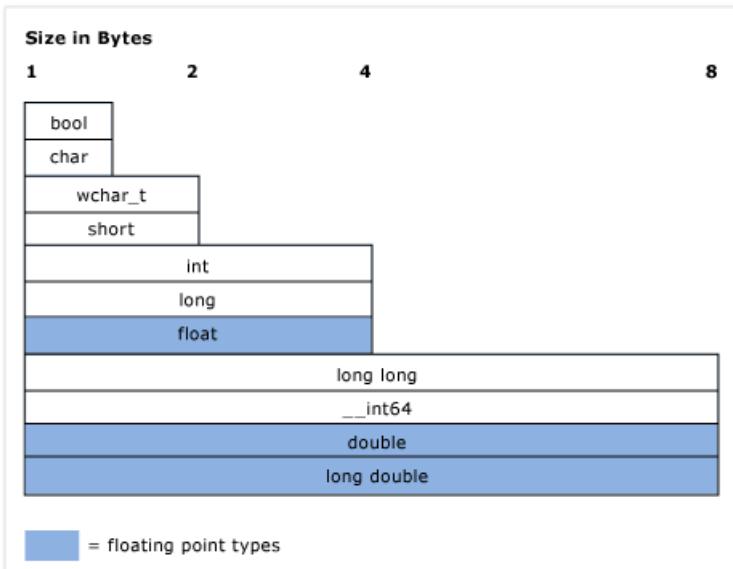
```
int result = 0;           // Declare and initialize an integer.
double coefficient = 10.8; // Declare and initialize a floating
                           // point value.
auto name = "Lady G. ";   // Declare a variable and let compiler
                           // deduce the type.
auto address;           // error. Compiler cannot deduce a type
                           // without an initializing value.
age = 12;                // error. Variable declaration must
                           // specify a type or use auto!
result = "Kenny G. ";    // error. Can't assign text to an int.
string result = "zero";  // error. Can't redefine a variable with
                           // new type.
int maxValue;            // Not recommended! maxValue contains
                           // garbage bits until it is initialized.
```

Tipos (internos) fundamentais

Ao contrário de algumas linguagens, C++ não tem tipo base universal do qual todos os outros tipos são derivados. O idioma inclui muitos *tipos fundamentais*, também conhecidos como *tipos internos*. Isso inclui tipos numéricos, como `int` `double` `long` `bool` mais os `char` `wchar_t` tipos e para caracteres ASCII e Unicode, respectivamente. Todos os tipos mais inteiros fundamentais (exceto `bool`, `double`, `wchar_t` e os tipos relacionados) têm `unsigned` versões, que modificam o intervalo de valores que a variável pode armazenar. Por exemplo, um `int`, que armazena um inteiro com sinal de 32 bits, pode representar um valor de -2.147.483.648 a 2.147.483.647. Um `unsigned int`, que também é armazenado como 32 bits, pode armazenar um valor de 0 a 4.294.967.295. O número total de valores possíveis em cada caso é o mesmo; somente o intervalo é diferente.

Os tipos fundamentais são reconhecidos pelo compilador, que tem regras internas que controlam que operações você poderá executar neles e como eles serão convertidos em outros tipos fundamentais. Para obter uma lista completa de tipos internos e seus limites de tamanho e numérico, consulte [tipos internos](#).

A ilustração a seguir mostra os tamanhos relativos dos tipos internos na implementação do Microsoft C++:



A tabela a seguir lista os tipos fundamentais usados com mais frequência e seus tamanhos na implementação do Microsoft C++:

TIPO	TAMANHO	COMENTÁRIO
<code>int</code>	4 bytes	A opção padrão para valores inteiros.
<code>double</code>	8 bytes	A opção padrão para valores de ponto flutuante.
<code>bool</code>	1 byte	Representa valores que podem ser <code>true</code> ou <code>false</code> .
<code>char</code>	1 byte	Use os caracteres ASCII em cadeias de caracteres do estilo C mais antigo ou objetos <code>std::string</code> que nunca precisarão ser convertidos em UNICODE.
<code>wchar_t</code>	2 bytes	Representa os valores de caractere "largos" que podem ser codificados no formato UNICODE (UTF-16 no Windows, outros sistemas operacionais podem ser diferentes). Esse é o tipo de caractere usado em cadeias de caracteres do tipo <code>std::wstring</code> .
<code>unsigned char</code>	1 byte	C++ não tem nenhum tipo de byte interno. Use <code>unsigned char</code> para representar um valor de byte.
<code>unsigned int</code>	4 bytes	Escolha padrão para sinalizadores de bit.
<code>long long</code>	8 bytes	Representa valores inteiros muito grandes.

Outras implementações de C++ podem usar tamanhos diferentes para determinados tipos numéricos. Para obter mais informações sobre os tamanhos e relações de tamanho que o padrão C++ exige, consulte [tipos internos](#).

O tipo `void`

O `void` tipo é um tipo especial; você não pode declarar uma variável do tipo `void`, mas pode declarar uma variável do tipo `void *` (ponteiro para `void`), que às vezes é necessário ao alocar memória bruta (não tipada). No entanto, os ponteiros para `void` não são de tipo seguro e, em geral, seu uso é fortemente desencorajado no C++ moderno. Em uma declaração de função, um `void` valor de retorno significa que a função não retorna um valor; esse é um uso comum e aceitável do `void`. Enquanto a linguagem C exigia funções que têm zero parâmetros para declarar `void` na lista de parâmetros, por exemplo, `fou(void)` essa prática é desencorajada no C++ moderno e deve ser declarada `fou()`. Para obter mais informações, consulte [conversões de tipo e segurança de tipo](#).

Qualificador do tipo `const`

Qualquer tipo interno ou definido pelo usuário pode ser qualificado pela palavra-chave `const`. Além disso, as funções de membro podem ser `const` qualificadas e até mesmo `const` sobrearcrgadas. O valor de um `const` tipo não pode ser modificado depois que ele é inicializado.

```
const double PI = 3.1415;
PI = .75 //Error. Cannot modify const variable.
```

O `const` qualificador é usado extensivamente em declarações de função e variável, e a "correção constante" é um conceito importante em C++; basicamente, isso significa que `const`, no momento da compilação, esses valores não são modificados de forma não intencional. Para obter mais informações, consulte [const](#).

Um `const` tipo é diferente de sua versão não `const`; por exemplo, `const int` é um tipo distinto de `int`. Você pode usar o `const_cast` operador C++ nessas raras ocasiões em que você deve remover *const-qualidade* de uma variável. Para obter mais informações, consulte [conversões de tipo e segurança de tipo](#).

Tipos de cadeia de caracteres

Estritamente falando, a linguagem C++ não tem nenhum tipo de cadeia de caracteres interno; `char` e `wchar_t` armazenar caracteres únicos-você deve declarar uma matriz desses tipos para aproximar uma cadeia de caracteres, adicionando um valor nulo de encerramento (por exemplo, ASCII `'\0'`) ao elemento da matriz, um após o último caractere válido (também chamado de *cadeia de caracteres C-Style*). As cadeias de caracteres de estilo C exigiam que muito mais códigos fossem escritos ou o uso de funções da biblioteca de utilitários de cadeia de caracteres externos. Mas, no C++ moderno, temos os tipos de biblioteca padrão `std::string` (para cadeias de caracteres de tipo de 8 bits `char`) ou `std::wstring` (para `wchar_t` cadeias de caracteres de tipo de 16 bits). Esses contêineres de biblioteca do C++ Standard podem ser considerados como tipos de cadeia de caracteres nativos, pois fazem parte das bibliotecas padrão incluídas em qualquer ambiente de compilação C++ compatível. Basta usar a diretiva `#include <string>` para tornar esses tipos disponíveis em seu programa. (Se você estiver usando MFC ou ATL, a `CString` classe também estará disponível, mas não faz parte do padrão C++.) O uso de matrizes de caracteres com terminação nula (as cadeias de estilo C mencionadas anteriormente) é altamente desencorajado no C++ moderno.

Tipos definidos pelo usuário

Quando você define um `class`, `struct`, `union` ou `enum`, essa construção é usada no restante do seu código como se fosse um tipo fundamental. Ele tem um tamanho conhecido na memória e certas regras sobre como pode ser usado aplicado para verificar o tempo de compilação e, no runtime, para a vida útil de seu programa. As principais diferenças entre os tipos internos fundamentais e os tipos definidos pelo usuário são:

- O compilador não tem conhecimento interno de um tipo definido pelo usuário. Ele aprende o tipo quando ele encontra a definição pela primeira vez durante o processo de compilação.
- Você especifica que operações podem ser executadas em seu tipo, e como ele pode ser convertido em outros tipos, definindo (por meio de sobrecarga) os operadores apropriados, como membros de classe ou funções de não membro. Para obter mais informações, consulte [sobrecarga de função](#)

Tipos de ponteiro

Retorne às versões mais antigas da linguagem C, o C++ continua a declarar uma variável de um tipo de ponteiro usando o declarador especial `*` (asterisco). Um tipo de ponteiro armazena o endereço do local na memória em que o valor real de dados é armazenado. No C++ moderno, eles são chamados de *ponteiros brutos* e são acessados em seu código por meio de operadores especiais `*` (asterisco) ou `->` (traço com maior que). Isso é chamado de cancelamento de *referências* e qual deles você usa depende se você está desreferenciando um ponteiro para um escalar ou um ponteiro para um membro em um objeto. O trabalho com tipos de ponteiro foi durante muito tempo um dos aspectos mais desafiadores e confusos do desenvolvimento de programas em C e C++. Esta seção descreve alguns fatos e práticas para ajudar a usar ponteiros brutos se você quiser, mas no C++ moderno, ele não é mais necessário (ou recomendado) para usar ponteiros brutos para a propriedade do objeto, devido à

evolução do [ponteiro inteligente](#) (discutido mais no final desta seção). Ainda é útil e seguro usar ponteiros brutos para observar objetos, mas se você tiver de usá-los para a propriedade de objeto, faça isso com muito cuidado e considerando como os objetos possuídos por eles são criados e destruídos.

A primeira coisa que você deve saber é que a declaração de uma variável de ponteiro bruto alocará somente a memória necessária para armazenar um endereço do local da memória a que o ponteiro fará referência quando sua referência for removida. A alocação da memória do próprio valor de dados (também chamada de *armazenamento de backup*) ainda não está alocada. Ou seja, declarando uma variável de ponteiro bruto, você está criando uma variável do endereço de memória, não uma variável de dados real. Remover a referência de uma variável de ponteiro antes de verificar se ela contém um endereço válido para um repositório de backup causará um comportamento indefinido (geralmente um erro fatal) em seu programa. O exemplo a seguir demonstra esse tipo de erro:

```
int* pNumber;           // Declare a pointer-to-int variable.
*pNumber = 10;          // error. Although this may compile, it is
// a serious error. We are dereferencing an
// uninitialized pointer variable with no
// allocated memory to point to.
```

O exemplo remove a referência de um tipo de ponteiro sem ter memória alocada para armazenar os dados inteiros reais ou um endereço de memória válido atribuído a ele. O código a seguir corrige esses erros:

```
int number = 10;          // Declare and initialize a local integer
// variable for data backing store.
int* pNumber = &number;    // Declare and initialize a local integer
// pointer variable to a valid memory
// address to that backing store.
...
*pNumber = 41;            // Dereference and store a new value in
// the memory pointed to by
// pNumber, the integer variable called
// "number". Note "number" was changed, not
// "pNumber".
```

O exemplo de código corrigido usa a memória de pilha local para criar o repositório de backup para o qual `pNumber` aponta. Usamos um tipo fundamental para simplificar. Na prática, o armazenamento de backup para ponteiros é, geralmente, tipos definidos pelo usuário que são alocados dinamicamente em uma área de memória chamada *heap* (ou *armazenamento gratuito*) usando uma `new` expressão de palavra-chave (em programação em estilo C, a antiga `malloc()` função de biblioteca de tempo de execução C foi usada). Uma vez alocadas, essas variáveis são geralmente chamadas de objetos, especialmente se elas forem baseadas em uma definição de classe. A memória alocada com `new` deve ser excluída por uma `delete` instrução correspondente (ou, se você usou a `malloc()` função para alocá-la, a função de tempo de execução C `free()`).

No entanto, é fácil esquecer de excluir um objeto alocado dinamicamente, especialmente em um código complexo, que causa um bug de recurso chamado de *vazamento de memória*. Por esse motivo, o uso de ponteiros brutos é altamente desaconselhável em C++ moderno. É quase sempre melhor encapsular um ponteiro bruto em um [ponteiro inteligente](#), que liberará automaticamente a memória quando o destruidor for invocado (quando o código sair do escopo do ponteiro inteligente); usando ponteiros inteligentes, você praticamente elimina uma classe inteira de bugs em seus programas em C++. No exemplo a seguir, suponha que `MyClass` seja um tipo definido pelo usuário que tem um método público `DoSomeWork();`

```
void someFunction() {
    unique_ptr<MyClass> pMc(new MyClass);
    pMc->DoSomeWork();
}
// No memory leak. Out-of-scope automatically calls the destructor
// for the unique_ptr, freeing the resource.
```

Para obter mais informações sobre apontadores inteligentes, consulte [ponteiros inteligentes](#).

Para obter mais informações sobre conversões de ponteiro, consulte [conversões de tipo e segurança de tipo](#).

Para obter mais informações sobre ponteiros em geral, consulte [ponteiros](#).

Tipos de dados do Windows

Na programação clássica do Win32 para C e C++, a maioria das funções usa TYPEDEFs e `#define` macros específicos do Windows (definidos em `windef.h`) para especificar os tipos de parâmetros e valores de retorno. Esses tipos de dados do Windows são basicamente apenas nomes especiais (aliases) fornecidos para tipos internos de C/C++. Para obter uma lista completa desses TYPEDEFs e definições de pré-processador, consulte [tipos de dados do Windows](#). Alguns desses TYPEDEFs, como `HRESULT` e `LCID`, são úteis e descritivos. Outros, como `INT`, não têm nenhum significado especial e são apenas aliases para tipos de C++ fundamentais. Outros tipos de dados do Windows têm nomes que foram mantidos desde a época da programação em C e de processadores de 16 bits, e não têm finalidade ou significado em hardware ou sistemas operacionais modernos. Também há tipos de dados especiais associados à biblioteca de Windows Runtime, listados como [Windows Runtime tipos de dados base](#). Em C++ moderno, a orientação geral é dar preferência aos tipos C++ fundamentais, a menos que o tipo do Windows comunique qualquer significado adicional sobre como o valor deve ser interpretado.

Mais informações

Para obter mais informações sobre o sistema de tipos C++, consulte os tópicos a seguir.

[Tipos de valor](#)

Descreve *tipos de valor* juntamente com problemas relacionados ao seu uso.

[Conversões de tipo e segurança de tipo](#)

Descreve problemas de conversão de tipos comuns e mostra como evitá-los.

Consulte também

[Bem-vindo de volta ao C++](#)

[Referência da linguagem C++](#)

[Biblioteca padrão do C++](#)

Escopo (C++)

02/09/2020 • 8 minutes to read • [Edit Online](#)

Quando você declara um elemento Program, como uma classe, uma função ou uma variável, seu nome só pode ser "visto" e usado em determinadas partes do seu programa. O contexto no qual um nome é visível é chamado de seu *escopo*. Por exemplo, se você declarar uma variável `x` dentro de uma função, `x` só será visível dentro desse corpo de função. Ele tem *escopo local*. Você pode ter outras variáveis com o mesmo nome em seu programa; Contanto que estejam em escopos diferentes, eles não violam a regra de uma definição e nenhum erro é gerado.

Para variáveis não estáticas automáticas, o escopo também determina quando elas são criadas e destruídas na memória do programa.

Há seis tipos de escopo:

- **Escopo global** Um nome global é aquele declarado fora de qualquer classe, função ou namespace. No entanto, em C++ mesmo esses nomes existem com um namespace global implícito. O escopo de nomes globais se estende do ponto de declaração até o final do arquivo no qual eles são declarados. Para nomes globais, a visibilidade também é regida pelas regras de [ligação](#) que determinam se o nome está visível em outros arquivos no programa.
- **Escopo do namespace** Um nome que é declarado dentro de um [namespace](#), fora de qualquer definição de classe ou de enumeração ou bloco de função, é visível de seu ponto de declaração até o final do namespace. Um namespace pode ser definido em vários blocos em arquivos diferentes.
- **Escopo local** Um nome declarado em uma função ou lambda, incluindo os nomes de parâmetro, tem escopo local. Eles são geralmente chamados de "locais". Eles só ficam visíveis do ponto de declaração até o fim da função ou do corpo lambda. Escopo local é um tipo de escopo de bloco, que é discutido posteriormente neste artigo.
- **Escopo de classe** Os nomes dos membros da classe têm escopo de classe, que se estende por toda a definição de classe, independentemente do ponto de declaração. A acessibilidade de membros de classe é mais controlada pelas `public` `private` `protected` palavras-chave, e. Membros públicos ou protegidos só podem ser acessados usando os operadores de seleção de Membros (`.` or `->`) ou operadores de ponteiro para membro (`*` ou `->*`).
- **Escopo da instrução** Nomes declarados em uma `for` `if` instrução, `while` ou `switch` são visíveis até o final do bloco de instrução.
- **Escopo da função** Um [rótulo](#) tem um escopo de função, o que significa que ele é visível em todo um corpo de função, mesmo antes de seu ponto de declaração. O escopo da função torna possível escrever instruções como `goto cleanup` antes de o `cleanup` rótulo ser declarado.

Ocultando nomes

Você pode ocultar um nome declarando-o em um bloco fechado. Na figura a seguir, `i` é redeclarado dentro do bloco interno, ocultando assim a variável associada a `i` no escopo do bloco externo.

```

Sample::Func(char *szWhat)
{
    int i = 0;
    cout << "i = " << i << "\n";
    {
        int i = 7, j = 9;
        cout << "i = " << i << "\n"
            << "j = " << j << "\n";
    }
    cout << "i = " << i << "\n";
}

```

Outer block contains local-scope object i and format parameter szWhat.

Inner block contains local-scope objects i and j.

Ocultar o escopo e o nome do bloco

O resultado do programa mostrado na figura é:

```

i = 0
i = 7
j = 9
i = 0

```

NOTE

O argumento `szWhat` é considerado como presente no escopo da função. Portanto, ele será tratado como se tivesse sido declarado no bloco externo da função.

Ocultando nomes de classe

É possível ocultar nomes de classe declarando uma função, um objeto, uma variável ou um enumerador no mesmo escopo. No entanto, o nome da classe ainda pode ser acessado quando prefixado pela palavra-chave `class`.

```

// hiding_class_names.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

// Declare class Account at global scope.
class Account
{
public:
    Account( double InitialBalance )
        { balance = InitialBalance; }
    double GetBalance()
        { return balance; }
private:
    double balance;
};

double Account = 15.37;           // Hides class name Account

int main()
{
    class Account Checking( Account ); // Qualifies Account as
                                         // class name

    cout << "Opening account with a balance of: "
        << Checking.GetBalance() << "\n";
}
//Output: Opening account with a balance of: 15.37

```

NOTE

Qualquer lugar ao qual o nome de classe (`Account`) é chamado, a classe keyword deve ser usada para diferenciá-lo da conta variável com escopo global. Essa regra não se aplica quando o nome de classe ocorre no lado esquerdo do operador de resolução de escopo (`::`). Os nomes no lado esquerdo do operador de resolução de escopo são sempre considerados nomes de classe.

O exemplo a seguir demonstra como declarar um ponteiro para um objeto do tipo `Account` usando a `class` palavra-chave:

```
class Account *Checking = new class Account( Account );
```

O `Account` no inicializador (entre parênteses) na instrução anterior tem escopo global; é do tipo `double` .

NOTE

A reutilização de nomes de identificadores como mostrada neste exemplo é considerada um estilo de programação ruim.

Para obter informações sobre a declaração e a inicialização de objetos de classe, consulte [classes, estruturas e uniões](#). Para obter informações sobre como usar o `new` e `delete` operadores de armazenamento gratuito, consulte [operadores New e Delete](#).

Ocultando nomes com escopo global

Você pode ocultar nomes com escopo global declarando explicitamente o mesmo nome no escopo do bloco. No entanto, os nomes de escopo global podem ser acessados usando o operador de resolução de escopo (`::`).

```
#include <iostream>

int i = 7; // i has global scope, outside all blocks
using namespace std;

int main( int argc, char *argv[] ) {
    int i = 5; // i has block scope, hides i at global scope
    cout << "Block-scoped i has the value: " << i << "\n";
    cout << "Global-scoped i has the value: " << ::i << "\n";
}
```

```
Block-scoped i has the value: 5
Global-scoped i has the value: 7
```

Confira também

[Conceitos básicos](#)

Arquivos de cabeçalho (C++)

02/09/2020 • 8 minutes to read • [Edit Online](#)

Os nomes dos elementos do programa, como variáveis, funções, classes e assim por diante, devem ser declarados antes que possam ser usados. Por exemplo, você não pode apenas escrever `x = 42` sem primeiro declarar 'x'.

```
int x; // declaration
x = 42; // use x
```

A declaração informa ao compilador se o elemento é um, `int` uma `double` função, uma `class` ou alguma outra coisa. Além disso, cada nome deve ser declarado (direta ou indiretamente) em cada arquivo. cpp no qual é usado. Quando você compila um programa, cada arquivo. cpp é compilado de forma independente em uma unidade de compilação. O compilador não tem conhecimento de quais nomes são declarados em outras unidades de compilação. Isso significa que, se você definir uma classe ou função ou variável global, deverá fornecer uma declaração dessa coisa em cada arquivo. cpp adicional que a usa. Cada declaração dessa coisa deve ser exatamente idêntica em todos os arquivos. Uma ligeira inconsistência causará erros ou comportamento indesejado quando o vinculador tentar mesclar todas as unidades de compilação em um único programa.

Para minimizar o potencial de erros, o C++ adotou a Convenção de uso de *arquivos de cabeçalho* para conter declarações. Você faz as declarações em um arquivo de cabeçalho e, em seguida, usa a diretiva `#include` em cada arquivo. cpp ou outro arquivo de cabeçalho que requer essa declaração. A diretiva `#include` insere uma cópia do arquivo de cabeçalho diretamente no arquivo. cpp antes da compilação.

NOTE

No Visual Studio 2019, o recurso de *módulos* do C++ 20 é apresentado como uma melhoria e substituição eventual para arquivos de cabeçalho. Para obter mais informações, consulte [visão geral dos módulos em C++](#).

Exemplo

O exemplo a seguir mostra uma maneira comum de declarar uma classe e, em seguida, usá-la em um arquivo de origem diferente. Vamos começar com o arquivo de cabeçalho, `my_class.h`. Ele contém uma definição de classe, mas observe que a definição está incompleta; a função de membro `do_something` não está definida:

```
// my_class.h
namespace N
{
    class my_class
    {
        public:
            void do_something();
    };
}
```

Em seguida, crie um arquivo de implementação (normalmente com uma extensão. cpp ou semelhante). Vamos chamar o arquivo `my_class.cpp` e fornecer uma definição para a declaração de membro. Adicionamos uma `#include` diretiva para o arquivo "my_class. h" para que a declaração de `my_class` inserida neste ponto no arquivo. cpp e incluimos `<iostream>` para efetuar pull da declaração para `std::cout`. Observe que as aspas são usadas para arquivos de cabeçalho no mesmo diretório que o arquivo de origem, e os colchetes angulares são usados

para cabeçalhos de biblioteca padrão. Além disso, muitos cabeçalhos de biblioteca padrão não têm. h ou qualquer outra extensão de arquivo.

No arquivo de implementação, opcionalmente, podemos usar uma `using` instrução para evitar ter que qualificar cada menção de "my_class" ou "cout" com "N::" ou "std::". Não coloque `using` instruções em seus arquivos de cabeçalho!

```
// my_class.cpp
#include "my_class.h" // header in local directory
#include <iostream> // header in standard library

using namespace N;
using namespace std;

void my_class::do_something()
{
    cout << "Doing something!" << endl;
}
```

Agora, podemos usar `my_class` em outro arquivo. cpp. Podemos `#include` o arquivo de cabeçalho para que o compilador receba a declaração. Tudo o que o compilador precisa saber é que my_class é uma classe que tem uma função de membro público chamada `do_something()`.

```
// my_program.cpp
#include "my_class.h"

using namespace N;

int main()
{
    my_class mc;
    mc.do_something();
    return 0;
}
```

Depois que o compilador conclui a compilação de cada arquivo. cpp em arquivos. obj, ele passa os arquivos. obj para o vinculador. Quando o vinculador mescla os arquivos de objeto, ele encontra exatamente uma definição para my_class; Ele está no arquivo. obj produzido por my_class. cpp e a compilação é realizada com sucesso.

Incluir proteções

Normalmente, os arquivos de cabeçalho têm uma *proteção include* ou uma `#pragma once` diretiva para garantir que eles não sejam inseridos várias vezes em um único arquivo. cpp.

```
// my_class.h
#ifndef MY_CLASS_H // include guard
#define MY_CLASS_H

namespace N
{
    class my_class
    {
        public:
            void do_something();
    };
}

#endif /* MY_CLASS_H */
```

O que colocar em um arquivo de cabeçalho

Como um arquivo de cabeçalho pode potencialmente ser incluído por vários arquivos, ele não pode conter definições que possam produzir várias definições de mesmo nome. Os itens a seguir não são permitidos ou são considerados uma prática muito boa:

- definições de tipo interno no namespace ou escopo global
- definições de função não embutidas
- definições de variáveis não const
- definições de agregação
- namespaces sem nome
- usando diretivas

O uso da `using` diretiva não causará necessariamente um erro, mas pode potencialmente causar um problema, pois ele coloca o namespace no escopo em cada arquivo. cpp que inclui direta ou indiretamente esse cabeçalho.

Arquivo de cabeçalho de exemplo

O exemplo a seguir mostra os vários tipos de declarações e definições que são permitidas em um arquivo de cabeçalho:

```

// sample.h
#pragma once
#include <vector> // #include directive
#include <string>

namespace N // namespace declaration
{
    inline namespace P
    {
        //...
    }

    enum class colors : short { red, blue, purple, azure };

    const double PI = 3.14; // const and constexpr definitions
    constexpr int MeaningOfLife{ 42 };
    constexpr int get_meaning()
    {
        static_assert(MeaningOfLife == 42, "unexpected!"); // static_assert
        return MeaningOfLife;
    }
    using vstr = std::vector<int>; // type alias
    extern double d; // extern variable

#define LOG // macro definition

#ifndef LOG // conditional compilation directive
    void print_to_log();
#endif

    class my_class // regular class definition,
    { // but no non-inline function definitions

        friend class other_class;
    public:
        void do_something(); // definition in my_class.cpp
        inline void put_value(int i) { vals.push_back(i); } // inline OK

    private:
        vstr vals;
        int i;
    };

    struct RGB
    {
        short r{ 0 }; // member initialization
        short g{ 0 };
        short b{ 0 };
    };

    template <typename T> // template definition
    class value_store
    {
    public:
        value_store<T>() = default;
        void write_value(T val)
        {
            //... function definition OK in template
        }
    private:
        std::vector<T> vals;
    };

    template <typename T> // template declaration
    class value_widget;
}

}

```

Unidades de tradução e vinculação

02/09/2020 • 5 minutes to read • [Edit Online](#)

Em um programa C++, um *símbolo*, por exemplo, uma variável ou um nome de função, pode ser declarado qualquer número de vezes dentro de seu escopo, mas só pode ser definido uma vez. Essa regra é a "regra de definição única" (ODR). Uma *declaração* apresenta (ou reintroduz) um nome para o programa. Uma *definição* introduz um nome. Se o nome representar uma variável, uma definição a inicializará explicitamente. Uma *definição de função* consiste na assinatura mais o corpo da função. Uma definição de classe consiste no nome da classe seguido por um bloco que lista todos os membros da classe. (Os corpos de funções de membro podem, opcionalmente, ser definidos separadamente em outro arquivo.)

O exemplo a seguir mostra algumas declarações:

```
int i;
int f(int x);
class C;
```

O exemplo a seguir mostra algumas definições:

```
int i{42};
int f(int x){ return x * i; }
class C {
public:
    void DoSomething();
};
```

Um programa consiste em uma ou mais *unidades de tradução*. Uma unidade de tradução consiste em um arquivo de implementação e todos os cabeçalhos que ele inclui direta ou indiretamente. Os arquivos de implementação normalmente têm uma extensão de arquivo *cpp* ou *cxx*. Normalmente, os arquivos de cabeçalho têm uma extensão de *h* ou *HPP*. Cada unidade de tradução é compilada independentemente pelo compilador. Após a conclusão da compilação, o vinculador mescla as unidades de tradução compiladas em um único *programa*. As violações da regra ODR normalmente aparecem como erros do vinculador. Os erros do vinculador ocorrem quando o mesmo nome tem duas definições diferentes em unidades de tradução diferentes.

Em geral, a melhor maneira de tornar uma variável visível em vários arquivos é colocá-la em um arquivo de cabeçalho. Em seguida, adicione uma diretiva `#include` em cada arquivo *cpp* que exija a declaração. Ao adicionar as *proteções include* no conteúdo do cabeçalho, você garante que os nomes declarados sejam definidos apenas uma vez.

No C++ 20, os [módulos](#) são introduzidos como uma alternativa aprimorada para arquivos de cabeçalho.

Em alguns casos, pode ser necessário declarar uma variável global ou uma classe em um arquivo *cpp*. Nesses casos, você precisa de uma maneira de informar ao compilador e ao vinculador qual tipo de *ligação* o nome tem. O tipo de vínculo especifica se o nome do objeto se aplica apenas a um arquivo ou a todos os arquivos. O conceito de ligação se aplica somente a nomes globais. O conceito de ligação não se aplica a nomes declarados dentro de um escopo. Um escopo é especificado por um conjunto de chaves de circunscrição, como nas definições de função ou classe.

Vínculo externo vs. interno

Uma *função gratuita* é uma função que é definida em escopo global ou de namespace. Variáveis globais não

const e funções gratuitas por padrão têm *vínculo externo*; Eles são visíveis de qualquer unidade de tradução no programa. Portanto, nenhum outro objeto global pode ter esse nome. Um símbolo com *vínculo interno* ou *nenhum vínculo* é visível somente na unidade de tradução em que é declarado. Quando um nome tem ligação interna, o mesmo nome pode existir em outra unidade de tradução. Variáveis declaradas em definições de classe ou corpos de função não têm vínculos.

Você pode forçar um nome global a ter um vínculo interno declarando-o explicitamente como `static`. Isso limita sua visibilidade à mesma unidade de tradução na qual ela é declarada. Nesse contexto, `static` significa algo diferente quando aplicado a variáveis locais.

Os seguintes objetos têm vínculo interno por padrão:

- objetos const
- objetos constexpr
- typedefs
- objetos estáticos no escopo do namespace

Para dar uma ligação externa de objeto const, declare-a como `extern` e atribua um valor a ela:

```
extern const int value = 42;
```

Consulte [externo](#) para obter mais informações.

Confira também

[Conceitos básicos](#)

:::no-loc(main):::argumentos de linha de comando e de função

02/09/2020 • 13 minutes to read • [Edit Online](#)

Todos os programas C++ devem ter uma `:::no-loc(main):::` função. Se você tentar compilar um projeto C++ .exe sem uma `:::no-loc(main):::` função, o compilador gerará um erro. (Bibliotecas de vínculo dinâmico e `:::no-loc(static):::` bibliotecas não têm uma `:::no-loc(main):::` função.) A `:::no-loc(main):::` função é onde o código-fonte começa a execução, mas antes de um programa entrar na `:::no-loc(main):::` função, todos os `:::no-loc(static):::` membros da classe sem inicializadores explícitos são definidos como zero. No Microsoft C++, os `:::no-loc(static):::` objetos globais também são inicializados antes da entrada para `:::no-loc(main):::`. Várias restrições se aplicam à `:::no-loc(main):::` função que não se aplica a outras funções do C++. A `:::no-loc(main):::` função:

- Não pode ser sobre carregado (consulte [sobrecarga de função](#)).
- Não pode ser declarado como `:::no-loc(inline):::`.
- Não pode ser declarado como `:::no-loc(static):::`.
- Não pode ter seu endereço removido.
- Não pode ser chamada.

A `:::no-loc(main):::` função não tem uma declaração, pois ela está incorporada à linguagem. Se tiver feito isso, a sintaxe da declaração para `:::no-loc(main):::` ficaria assim:

```
int :::no-loc(main):::();  
int :::no-loc(main):::(int :::no-loc(argc):::, :::no-loc(char)::: *:::no-loc(argv):::[], :::no-loc(char):::  
*:::no-loc(envp):::[]);
```

Específico da Microsoft

Se os arquivos de origem usarem acters de largura Unicode `:::no-loc(char):::`, você poderá usar `:::no-loc(wmain):::` o, que é a `:::no-loc(char):::` versão de grande acter do `:::no-loc(main):::`. A sintaxe de declaração para `:::no-loc(wmain):::` é a seguinte:

```
int :::no-loc(wmain):::( );  
int :::no-loc(wmain):::(int :::no-loc(argc):::, :::no-loc(wchar_t)::: *:::no-loc(argv):::[], :::no-  
loc(wchar_t)::: *:::no-loc(envp):::[]);
```

Você também pode usar `:::no-loc(_tmain):::`, que é definido em `t :::no-loc(char):::.h`. `:::no-loc(_tmain):::` resolve para a `:::no-loc(main):::` menos que `_UNICODE` seja definido. Nesse caso, `:::no-loc(_tmain):::` resolve a `:::no-loc(wmain):::`.

Se nenhum valor de retorno for especificado, o compilador fornecerá um valor de retorno igual a zero. Como alternativa, as `:::no-loc(main):::` `:::no-loc(wmain):::` funções e podem ser declaradas como retornando `:::no-loc(void):::` (nenhum valor de retorno). Se você declarar `:::no-loc(main):::` ou `:::no-loc(wmain):::` retornar `:::no-loc(void):::`, não poderá retornar um `:::no-loc(exit):::` código para o processo pai ou sistema operacional usando uma instrução `Return`. Para retornar um `:::no-loc(exit):::` código quando `:::no-loc(main):::` ou `:::no-loc(wmain):::` é declarado como `:::no-loc(void):::`, você deve usar a `:::no-loc(exit):::` função.

FINAL específico da Microsoft

Argumentos de linha de comando

Os argumentos para `::::no-loc(main):::` ou `::::no-loc(wmain):::` permitem a análise de linha de comando conveniente de argumentos e, opcionalmente, o acesso a variáveis de ambiente. Os tipos para `::::no-loc(argc):::` e `::::no-loc(argv):::` são definidos pela linguagem. Os nomes `::::no-loc(argc):::`, `::::no-loc(argv):::` e `::::no-loc(envp):::` são tradicionais, mas você pode nomeá-los como desejar.

```
int ::::no-loc(main):::( int ::::no-loc(argc):::, ::::no-loc(char):::* ::::no-loc(argv):::[], ::::no-loc(char):::* ::::no-loc(envp):::[]);  
int ::::no-loc(wmain):::( int ::::no-loc(argc):::, ::::no-loc(wchar_t):::* ::::no-loc(argv):::[], ::::no-loc(wchar_t):::* ::::no-loc(envp):::[]);
```

As definições dos argumentos são as seguintes:

`::::no-loc(argc):::`

Um inteiro que contém a contagem de argumentos que acompanham `::::no-loc(argv):::`. O `::::no-loc(argc):::` parâmetro é sempre maior ou igual a 1.

`::::no-loc(argv):::`

Uma matriz de cadeias de caracteres terminadas em nulo que representam argumentos de linha de comando inseridos pelo usuário do programa. Por convenção, `::::no-loc(argv):::[0]` é o comando com o qual o programa é invocado, `::::no-loc(argv):::[1]` é o primeiro argumento de linha de comando e assim por diante, `::::no-loc(argv):::[::::no-loc(argc):::]` que é sempre nulo. Consulte [Personalizando o processamento de linha de comando](#) para obter informações sobre como suprimir o processamento de linha de comando.

O primeiro argumento de linha de comando sempre é `::::no-loc(argv):::[1]` e o último é `::::no-loc(argv):::[::::no-loc(argc)::: - 1]`.

NOTE

Por convenção, `::::no-loc(argv):::[0]` é o comando com o qual o programa é invocado. No entanto, é possível gerar um processo usando `::::no-loc(CreateProcess):::` e se você usar o primeiro e o segundo argumentos (*LpApplicationName e lpCommandLine*), `::::no-loc(argv):::[0]` talvez não seja o nome do executável; use `::::no-loc(GetModuleFileName):::` para recuperar o nome do executável e seu caminho totalmente qualificado.

Específico da Microsoft

`::::no-loc(envp):::`

A `::::no-loc(envp):::` matriz, que é uma extensão comum em muitos sistemas UNIX, é usada no Microsoft C++. Trata-se de uma matriz de cadeias de caracteres que representam as variáveis definidas no ambiente do usuário. Essa matriz é encerrada por uma entrada NULL. Ele pode ser declarado como uma matriz de ponteiros para `::::no-loc(char)::: (::::no-loc(char)::: * ::::no-loc(envp):::[])` ou como um ponteiro para ponteiros para `::::no-loc(char)::: (::::no-loc(char)::: * * ::::no-loc(envp):::[])`. Se o seu programa usar `::::no-loc(wmain):::` em vez de `::::no-loc(main):::`, use o `::::no-loc(wchar_t):::` tipo de dados em vez de `::::no-loc(char):::`. O bloco de ambiente passou para `::::no-loc(main):::` e `::::no-loc(wmain):::` é uma cópia "congelada" do ambiente atual. Se você alterar posteriormente o ambiente por meio de uma chamada para `putenv` ou `_wputenv`, o ambiente atual (como retornado por `getenv` ou `_wgetenv` e a `_environ` `_wenviron` variável or) será alterado, mas o bloco apontado por `::::no-loc(envp):::` não será alterado. Consulte [Personalizando o processamento de linha de comando](#) para obter informações sobre como suprimir o processamento do ambiente. Esse argumento é compatível com ANSI em C, mas não em C++.

FINAL específico da Microsoft

Exemplo

O exemplo a seguir mostra como usar os `:::no-loc(argc):::` `:::no-loc(argv):::` argumentos, e `:::no-loc(envp):::` para

```
:::no-loc(main):::
```

```
// argument_definitions.cpp
// compile with: /EHsc
#include <iostream>
#include <string.h>

using namespace std;
int :::no-loc(main):::( int :::no-loc(argc):::, :::no-loc(char)::: *:::no-loc(argv):::[], :::no-loc(char):::
*:::no-loc(envp):::[ ] ) {
    int iNumberLines = 0;    // Default is no line numbers.

    // If /n is passed to the .exe, display numbered listing
    // of environment variables.

    if ( ( :::no-loc(argc)::: == 2) && _stricmp( :::no-loc(argv):::[1], "/n" ) == 0 )
        iNumberLines = 1;

    // Walk through list of strings until a NULL is encountered.
    for( int i = 0; :::no-loc(envp):::[i] != NULL; ++i ) {
        if( iNumberLines )
            cout << i << ":" << :::no-loc(envp):::[i] << "\n";
    }
}
```

Analizando argumentos de linha de comando C++

Específico da Microsoft

O código de inicialização do Microsoft C/C++ usa as seguintes regras para interpretar os argumentos dados na linha de comando do sistema operacional:

- Os argumentos são delimitados por espaço em branco, que é um espaço ou uma tabulação.
- O cursor `:::no-loc(char):::` acter (^) não é reconhecido como um acter de escape `:::no-loc(char):::` ou delimitador. O `:::no-loc(char):::` acter é manipulado completamente pelo analisador de linha de comando no sistema operacional antes de ser passado para `:::no-loc(argv):::` a matriz no programa.
- Uma cadeia de caracteres entre aspas duplas ("String") é interpretada como um único argumento, independentemente do espaço em branco contido em. Uma cadeia de caracteres entre aspas pode ser inserida em um argumento.
- Aspas duplas precedidas por uma barra invertida (\ ") é interpretada como uma aspa dupla literal `:::no-loc(char):::` acter ("").
- As barras invertidas são interpretadas literalmente, a menos que precedam imediatamente as aspas duplas.
- Se um número de barras invertidas mesmo for seguido por aspas duplas, uma barra invertida é colocada na matriz de `:::no-loc(argv):::` para cada par de barras invertidas, e aspas duplas são interpretadas como um delimitador de cadeia de caracteres.
- Se um número ímpar de barras invertidas for seguido por uma aspa dupla, uma barra invertida será colocada na `:::no-loc(argv):::` matriz para cada par de barras invertidas e as aspas duplas serão "escapadas" pela `:::no-loc(main):::` barra invertida, fazendo com que uma aspa dupla literal ("") seja colocada em `:::no-loc(argv):::`.

Exemplo

O programa a seguir demonstra como argumentos de linha de comando são passados:

```

// command_line_arguments.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
int ::::no-loc(main):::( int ::::no-loc(argc):::, // Number of strings in array ::::no-loc(argv):::
                           ::::no-loc(char)::: *:::no-loc(argv):::[], // Array of command-line argument strings
                           ::::no-loc(char)::: *:::no-loc(envp):::[ ] ) // Array of environment variable strings
{
    int count;

    // Display each command-line argument.
    cout << "\nCommand-line arguments:\n";
    for( count = 0; count < ::::no-loc(argc):::; count++ )
        cout << " ::::no-loc(argv):::[ " << count << " ]   "
            << ::::no-loc(argv):::[count] << "\n";
}

```

A tabela a seguir mostra exemplos de entrada e saídas esperadas, demonstrando as regras na lista acima.

Resultados da análise de linhas de comando

ENTRADA DE LINHA DE COMANDO	:::NO-LOC(ARGV):::[1]	:::NO-LOC(ARGV):::[2]	:::NO-LOC(ARGV):::BETA
"abc" d e	abc	d	e
a\\b d"e f"g h	a\\b	de fg	h
a\\\"b c d	a\"b	c	d
a\\\\\"b c" d e	a\\\"b c	d	e

FINAL específico da Microsoft

Expansão de curinga

Específico da Microsoft

Você pode usar caracteres curinga — o ponto de interrogação (?) e o asterisco (*) — para especificar argumentos de nome de arquivo e caminho na linha de comando.

Os argumentos de linha de comando são tratados por uma rotina chamada `_set:::no-loc(argv):::` (ou `_wset:::no-loc(argv):::`) no ambiente de Wide-`:::no-loc(char):::` acter), que, por padrão, não expande curingas em cadeias de caracteres separadas na `:::no-loc(argv):::` matriz de cadeias. Para obter mais informações sobre como habilitar a expansão de curinga, consulte [expandindo argumentos curinga](#).

FINAL específico da Microsoft

Personalizando processamento de linha de comando C++

Específico da Microsoft

Se o seu programa não utiliza argumentos de linha de comando, é possível economizar um pequeno espaço suprimindo o uso da rotina de biblioteca que executa o processamento de linha de comando. Essa rotina é chamada `_set:::no-loc(argv):::` e é descrita na [expansão de curinga](#). Para suprimir seu uso, defina uma rotina que não faz nada no arquivo que contém a `:::no-loc(main):::` função e nomeie-o `_set:::no-loc(argv):::`. A chamada para `_set:::no-loc(argv):::` é então satisfeita pela sua definição de `_set:::no-loc(argv):::` e a versão

da biblioteca não é carregada.

Da mesma forma, se você nunca acessar a tabela de ambiente por meio do `:::no-loc(envp):::` argumento, poderá fornecer sua própria rotina vazia a ser usada no lugar da `_set:::no-loc(envp):::` rotina de processamento de ambiente. Assim como com a `_set:::no-loc(argv):::` função, `_set:::no-loc(envp):::` deve ser declarado como `** ::no-loc(extern):: "C"**`.

Seu programa pode fazer chamadas para a `spawn` `exec` família de rotinas ou na biblioteca de tempo de execução do C. Se tiver, você não deverá suprimir a rotina de processamento de ambiente, pois essa rotina é usada para passar um ambiente do processo pai para o processo filho.

FINAL específico da Microsoft

Confira também

[Conceitos básicos](#)

Encerramento do programa C++

02/09/2020 • 4 minutes to read • [Edit Online](#)

Em C++, você pode ::no-loc(exit)::: um programa das seguintes maneiras:

- Chame a função.
- Chame a função.
- Execute uma instrução de ::no-loc(main):::.

Função ::no-loc(exit):::

A ::no-loc(exit)::: função, declarada em `<stdlib.h>`, encerra um programa em C++. O valor fornecido como um argumento para ::no-loc(exit)::: é ::no-loc(return)::: Ed para o sistema operacional como ::no-loc(return)::: código ou código do programa ::no-loc(exit):::. Por convenção, um ::no-loc(return)::: código zero significa que o programa foi concluído com êxito. Você pode usar as constantes `EXIT_FAILURE` e `EXIT_SUCCESS`, também definidas no `<stdlib.h>`, para indicar o êxito ou a falha do seu programa.

A emissão de uma ::no-loc(return)::: instrução da ::no-loc(main)::: função é equivalente à chamada da ::no-loc(exit)::: função com o ::no-loc(return)::: valor como seu argumento.

Função ::no-loc(abort):::

A ::no-loc(abort)::: função, também declarada no arquivo de inclusão padrão `<stdlib.h>`, encerra um programa em C++. A diferença entre o ::no-loc(exit)::: e ::no-loc(abort)::: é que ::no-loc(exit)::: permite que o processamento de encerramento de tempo de execução do C++ ocorra (destruidores de objeto global serão chamados), enquanto ::no-loc(abort)::: encerra o programa imediatamente. A ::no-loc(abort)::: função ignora o processo de destruição normal para objetos estáticos globais inicializados. Ele também ignora qualquer processamento especial que foi especificado usando a ::no-loc(atexit)::: função.

Função ::no-loc(atexit):::

Use a ::no-loc(atexit)::: função para especificar ações que são executadas antes do encerramento do programa. Nenhum objeto estático global inicializado antes da chamada para ::no-loc(atexit)::: é destruído antes da execução da ::no-loc(exit)::: função de processamento.

::no-loc(return):::instrução em ::no-loc(main):::

A emissão de uma instrução de ::no-loc(main)::: é funcionalmente equivalente à chamada da ::no-loc(exit)::: função. Considere o exemplo a seguir:

```
// ::no-loc(return):::_statement.cpp
#include <stdlib.h>
int ::no-loc(main):::( )
{
    ::no-loc(exit):::( 3 );
    ::no-loc(return)::: 3;
}
```

As ::no-loc(exit)::: ::no-loc(return)::: instruções e no exemplo anterior são funcionalmente idênticas. No entanto, o C++ requer que funções com ::no-loc(return)::: tipos diferentes de ::no-loc(void)::: ::no-loc(return):::

um valor. A `::::no-loc(return):::` instrução permite que você tenha `::::no-loc(return):::` um valor de `::::no-loc(main):::`.

Destruicão de objetos estáticos

Quando você chama `::::no-loc(exit):::` ou executa uma `::::no-loc(return):::` instrução de `::::no-loc(main):::`, os objetos estáticos são destruídos na ordem inversa de sua inicialização (após a chamada para `::::no-loc(atexit):::` se houver). O exemplo a seguir mostra como realizar essa inicialização e as tarefas de limpeza.

Exemplo

No exemplo a seguir, os objetos estáticos `sd1` e `sd2` são criados e inicializados antes da entrada para `::::no-loc(main):::`. Depois que este programa termina usando a `::::no-loc(return):::` instrução, o primeiro `sd2` é destruído e, em seguida, `sd1`. O destruidor da classe `ShowData` fecha os arquivos associados a esses objetos estáticos.

```
// using_:::no-loc(exit):::_or_:::no-loc(return):::1.cpp
#include <stdio.h>
class ShowData {
public:
    // Constructor opens a file.
    ShowData( const char *szDev ) {
        errno_t err;
        err = fopen_s(&OutputDev, szDev, "w" );
    }

    // Destructor closes the file.
    ~ShowData() { fclose( OutputDev ); }

    // Disp function shows a string on the output device.
    ::::no-loc(void)::: Disp( char *szData ) {
        fputs( szData, OutputDev );
    }
private:
    FILE *OutputDev;
};

// Define a static object of type ShowData. The output device
// selected is "CON" -- the standard output device.
ShowData sd1 = "CON";

// Define another static object of type ShowData. The output
// is directed to a file called "HELLO.DAT"
ShowData sd2 = "hello.dat";

int ::::no-loc(main):::() {
    sd1.Disp( "hello to default device\n" );
    sd2.Disp( "hello to file hello.dat\n" );
}
```

Outra maneira de escrever o código é declarar os objetos `ShowData` com escopo de bloco, permitindo que eles sejam destruídos quando saírem do escopo:

```
int ::::no-loc(main):::() {
    ShowData sd1, sd2( "hello.dat" );

    sd1.Disp( "hello to default device\n" );
    sd2.Disp( "hello to file hello.dat\n" );
}
```

Confira também

www.observatoriodocinema.com.br

Lvalues e Rvalues (C++)

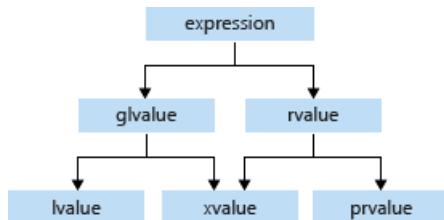
02/09/2020 • 4 minutes to read • [Edit Online](#)

Cada expressão C++ tem um tipo e pertence a uma *categoria de valor*. As categorias de valor são a base para as regras que os compiladores devem seguir ao criar, copiar e mover objetos temporários durante a avaliação da expressão.

O padrão C++ 17 define as categorias de valor de expressão da seguinte maneira:

- Um *glvalue* é uma expressão cuja avaliação determina a identidade de um objeto, um campo de bits ou uma função.
- Um *não prvalue* é uma expressão cuja avaliação inicializa um objeto ou um campo de bits, ou computa o valor do operando de um operador, conforme especificado pelo contexto no qual ele aparece.
- Um *xValue* é um glvalue que denota um objeto ou um campo de bits cujos recursos podem ser reutilizados (geralmente porque está próximo ao final do seu tempo de vida). Exemplo: determinados tipos de expressões que envolvem referências a rvalue (8.3.2) produzem XValues, como uma chamada para uma função cujo tipo de retorno é uma referência rvalue ou uma conversão para um tipo de referência rvalue.
- Um *lvalue* é um glvalue que não é um xValue.
- Um *rvalue* é um não prvalue ou um xValue.

O diagrama a seguir ilustra as relações entre as categorias:



Um lvalue tem um endereço que seu programa pode acessar. Exemplos de expressões lvalue incluem nomes de variáveis, incluindo `const` variáveis, elementos de matriz, chamadas de função que retornam uma referência de lvalue, campos de bits, uniões e membros de classe.

Uma expressão não prvalue não tem endereço acessível por seu programa. Exemplos de expressões não prvalue incluem literais, chamadas de função que retornam um tipo de não referência e objetos temporários que são criados durante a expressão evalution, mas acessíveis somente pelo compilador.

Uma expressão xValue tem um endereço que não é mais acessível pelo seu programa, mas pode ser usada para inicializar uma referência rvalue, que fornece acesso à expressão. Os exemplos incluem chamadas de função que retornam uma referência rvalue e o subscrito, o membro e o ponteiro da matriz para expressões de membro em que a matriz ou o objeto é uma referência rvalue.

Exemplo

O exemplo a seguir demonstra vários usos corretos e incorretos de l-values e r-values:

```
// lvalues_and_rvalues2.cpp
int main()
{
    int i, j, *p;

    // Correct usage: the variable i is an lvalue and the literal 7 is a prvalue.
    i = 7;

    // Incorrect usage: The left operand must be an lvalue (C2106).`j * 4` is a prvalue.
    7 = i; // C2106
    j * 4 = 7; // C2106

    // Correct usage: the dereferenced pointer is an lvalue.
    *p = i;

    // Correct usage: the conditional operator returns an lvalue.
    ((i < 3) ? i : j) = 7;

    // Incorrect usage: the constant ci is a non-modifiable lvalue (C3892).
    const int ci = 7;
    ci = 9; // C3892
}
```

NOTE

Os exemplos neste tópico ilustram o uso correto e incorreto quando os operadores não são sobrecarregados. Ao sobrecarregar os operadores, é possível tornar uma expressão como `j * 4` um l-value.

Os termos *lvalue* e *rvalue* geralmente são usados quando você faz referência a referências de objeto. Para obter mais informações sobre referências, consulte [Declarador de referência Lvalue: &](#) e o [Declarador de referência rvalue: &&](#).

Confira também

[Conceitos básicos](#)

[Declarador de referência Lvalue: &](#)

[Declarador de referência Rvalue: &&](#)

Objetos temporários

02/09/2020 • 3 minutes to read • [Edit Online](#)

Em alguns casos, é necessário que o compilador crie objetos temporários. Esses objetos temporários podem ser criados pelos seguintes motivos:

- Para inicializar uma `const` referência com um inicializador de um tipo diferente daquele do tipo subjacente da referência que está sendo inicializada.
- Para armazenar o valor de retorno de uma função que retorna um tipo definido pelo usuário. Esses temporários só serão criados quando o programa não copia o valor de retorno para um objeto. Por exemplo:

```
UDT Func1();    // Declare a function that returns a user-defined
                // type.

...
Func1();        // Call Func1, but discard return value.
                // A temporary object is created to store the return
                // value.
```

Como o valor de retorno não é copiado a outro objeto, um objeto temporário será criado. Um caso comum mais comum onde os temporários são criados é durante a avaliação de uma expressão onde as funções sobrecarregadas do operador devem ser chamadas. Essas funções sobrecarregadas do operador retornam um tipo definido pelo usuário que geralmente não é copiado a outro objeto.

Considere a expressão `ComplexResult = Complex1 + Complex2 + Complex3`. A expressão `Complex1 + Complex2` é avaliada, e o resultado é armazenado em um objeto temporário. Em seguida, a expressão *temporária* `+ Complex3` é avaliada e o resultado é copiado para `ComplexResult` (supondo que o operador de atribuição não esteja sobrecarregado).

- Para armazenar o resultado de uma conversão em um tipo definido pelo usuário. Quando um objeto de um determinado tipo é convertido explicitamente em um tipo definido pelo usuário, o novo objeto é criado como um objeto temporário.

Os objetos temporários têm um tempo de vida definido pelo ponto de criação e pelo ponto em que são destruídos. Qualquer expressão que cria mais de um objeto temporário acaba destruindo-os na ordem inversa da qual foram criados. Os pontos em que a destruição ocorre são mostrados na tabela a seguir.

Pontos de destruição para objetos temporários

MOTIVO TEMPORÁRIO CRIADO	PONTO DE DESTRUÇÃO
Resultado da avaliação de expressão	Todos os temporaries criados como resultado da avaliação de expressão são destruídos no final da instrução de expressão (ou seja, no ponto-e-vírgula) ou no final das expressões de controle para <code>for</code> <code>if</code> instruções, <code>while</code> , <code>do</code> e <code>switch</code> .

MOTIVO TEMPORÁRIO CRIADO	PONTO DE DESTRUÇÃO
Inicializando <code>const</code> referências	<p>Se um inicializador não for um valor <code>l</code> do mesmo tipo da referência que está sendo inicializada, um temporário do tipo de objeto subjacente será criada e inicializada com a expressão de inicialização. Esse objeto temporário será destruído imediatamente depois que o objeto de referência ao qual está associado é destruído.</p>

Alinhamento

02/09/2020 • 7 minutes to read • [Edit Online](#)

Um dos recursos de baixo nível do C++ é a capacidade de especificar o alinhamento preciso de objetos na memória para tirar o máximo proveito de uma arquitetura de hardware específica. Por padrão, o compilador alinha os membros de classe e struct em seu valor de tamanho: `bool` e `char` em limites de 1 byte, `short` em limites de 2 bytes, `int`, `long`, e `float` em limites de 4 bytes, e `long long`, `double` e `long double` em limites de 8 bytes.

Na maioria dos cenários, você nunca precisa se preocupar com alinhamento porque o alinhamento padrão já é o ideal. Em alguns casos, no entanto, você pode obter melhorias significativas de desempenho ou economia de memória, especificando um alinhamento personalizado para suas estruturas de dados. Antes do Visual Studio 2015, você pode usar as palavras-chave específicas da Microsoft `__alignof` e `__declspec(align)` especificar um alinhamento maior que o padrão. A partir do Visual Studio 2015, você deve usar as palavras-chave padrão do C++ 11 `alignof` e `alignas` para portabilidade de código máxima. As novas palavras-chave se comportam da mesma forma nos bastidores que as extensões específicas da Microsoft. A documentação para essas extensões também se aplica às novas palavras-chave. Para obter mais informações, consulte [alignof operador](#) e [alinear](#). O padrão C++ não especifica o comportamento de empacotamento para alinhamento em limites menores do que o padrão do compilador para a plataforma de destino, para que você ainda precise usar a Microsoft `#pragma pack` nesse caso.

Use a [classe aligned_storage](#) para a alocação de memória de estruturas de dados com alinhamentos personalizados. A [classe aligned_union](#) é para especificar o alinhamento de uniões com construtores ou destruidores não triviais.

Alinhamento e endereços de memória

O alinhamento é uma propriedade de um endereço de memória, expressa como o módulo de endereço numérico a potência de 2. Por exemplo, o endereço 0x0001103F módulo 4 é 3. Esse endereço é considerado alinhado com $4n + 3$, em que 4 indica a potência escolhida de 2. O alinhamento de um endereço depende da potência escolhida 2. O mesmo módulo de endereço 8 é 7. Um endereço será considerado alinhado ao X se seu alinhamento for $xn + 0$.

As CPUs executam instruções que operam em dados armazenados na memória. Os dados são identificados por seus endereços na memória. Uma única referência também tem um tamanho. Chamamos uma referência *naturalmente alinhada* se seu endereço estiver alinhado ao seu tamanho. Caso contrário, ele será chamado *desalinhado*. Por exemplo, uma referência de ponto flutuante de 8 bytes será naturalmente alinhada se o endereço usado para identificá-lo tiver um alinhamento de 8 bytes.

Manipulação de alinhamento de dados do compilador

Os compiladores tentam fazer alocações de dados de forma a impedir o desalinhamento dos dados.

Para tipos de dados simples, o compilador atribui endereços que são múltiplos do tamanho em bytes do tipo de dados. Por exemplo, o compilador atribui endereços a variáveis do tipo `long` que são múltiplos de 4, definindo os dois bits inferiores do endereço como zero.

O compilador também ajusta as estruturas de uma forma que alinha naturalmente cada elemento da estrutura. Considere a estrutura `struct x_` no exemplo de código a seguir:

```
struct x_
{
    char a;      // 1 byte
    int b;       // 4 bytes
    short c;    // 2 bytes
    char d;      // 1 byte
} bar[3];
```

O compilador pads essa estrutura para impor o alinhamento naturalmente.

O exemplo de código a seguir mostra como o compilador coloca a estrutura preenchida na memória:

```
// Shows the actual memory layout
struct x_
{
    char a;          // 1 byte
    char _pad0[3];  // padding to put 'b' on 4-byte boundary
    int b;          // 4 bytes
    short c;        // 2 bytes
    char d;          // 1 byte
    char _pad1[1];  // padding to make sizeof(x_) multiple of 4
} bar[3];
```

Ambas as declarações retornam `sizeof(struct x_)` como 12 bytes.

A segunda declaração inclui dois elementos de preenchimento:

1. `char _pad0[3]` para alinhar o `int b` membro em um limite de 4 bytes.
2. `char _pad1[1]` para alinhar os elementos de matriz da estrutura `struct _x bar[3];` em um limite de quatro bytes.

O preenchimento alinha os elementos de `bar[3]` uma maneira que permite o acesso natural.

O exemplo de código a seguir mostra o `bar[3]` layout da matriz:

adr	offset	element
0x0000		char a; // bar[0]
0x0001		char _pad0[3];
0x0004		int b;
0x0008		short c;
0x000a		char d;
0x000b		char _pad1[1];
0x000c		char a; // bar[1]
0x000d		char _pad0[3];
0x0010		int b;
0x0014		short c;
0x0016		char d;
0x0017		char _pad1[1];
0x0018		char a; // bar[2]
0x0019		char _pad0[3];
0x001c		int b;
0x0020		short c;
0x0022		char d;
0x0023		char _pad1[1];

`alignof` e `alignas`

O `alignas` especificador de tipo é uma maneira portátil e C++ padrão de especificar o alinhamento personalizado de variáveis e tipos definidos pelo usuário. O `alignof` operador é, da mesma forma, uma maneira portátil e padrão de obter o alinhamento de um tipo ou variável especificada.

Exemplo

Você pode usar `alignas` em uma classe, struct ou Union ou em membros individuais. Quando vários `alignas` especificadores forem encontrados, o compilador escolherá o mais estrito (aquele com o maior valor).

```
// alignas_alignof.cpp
// compile with: cl /EHsc alignas_alignof.cpp
#include <iostream>

struct alignas(16) Bar
{
    int i;          // 4 bytes
    int n;          // 4 bytes
    alignas(4) char arr[3];
    short s;        // 2 bytes
};

int main()
{
    std::cout << alignof(Bar) << std::endl; // output: 16
}
```

Confira também

[Alinhamento da estrutura de dados](#)

Tipos literal, POD, layout padrão e trivial

22/04/2020 • 10 minutes to read • [Edit Online](#)

O termo *layout* se refere a como os membros de um objeto de classe, struct ou tipo de união são organizados na memória. Em alguns casos, o layout é bem definido pela especificação da linguagem de programação. Mas, quando uma classe ou struct contém determinados recursos da linguagem de programação C++, como classes base virtuais, funções virtuais, membros com controle de acesso diferente, o compilador escolhe um layout. Esse layout pode variar dependendo de quais otimizações estão sendo executadas e, em muitos casos, o objeto pode não ocupar uma área contígua da memória. Por exemplo, se uma classe tem funções virtuais, todas as instâncias dessa classe podem compartilhar uma única tabela de função virtual. Esses tipos são muito úteis, mas também têm limitações. Como o layout é indefinido, eles não podem ser passados para os programas escritos em outras linguagens de programação, como C, e como podem ser não contíguos, não é possível copiá-los de modo confiável com funções de nível inferior rápidas, como `memcpy`, ou serializados em uma rede.

Para habilitar os compiladores, bem como programas e metaprogramas C++ para ponderar sobre a adequação de um determinado tipo para operações que dependem de um layout de memória específico, o C++14 introduziu três categorias de classes e structs simples: *trivial*, *layout padrão* e *POD* (dados antigos simples). A Biblioteca Padrão tem os modelos de função `is_trivial<T>`, `is_standard_layout<T>` e `is_pod<T>` que determinam se um tipo específico pertence a uma determinada categoria.

Tipos triviais

Quando uma classe ou struct no C++ tem funções de membro especial explicitamente padronizadas ou fornecidas pelo compilador, trata-se de um tipo trivial. Esse tipo ocupa uma área de memória contígua. Ele pode ter membros com especificadores de acesso diferentes. No C++, o compilador escolhe como ordenar os membros nessa situação. Portanto, você pode copiar esses objetos na memória, mas não pode consumi-los confiavelmente de um programa em C. Um tipo trivial T pode ser copiado em uma matriz char ou unsigned char e copiado novamente, com segurança, em uma variável T. Observe que, devido aos requisitos de alinhamento, pode haver bytes de preenchimento entre os membros de tipo.

Tipos triviais têm um construtor padrão trivial, um construtor de cópia trivial, um operador de atribuição de cópia trivial e um destruidor trivial. Em cada caso, *trivial* significa que o construtor/operador/destruidor não é fornecido pelo usuário e pertence a uma classe que

- não tem funções virtuais nem classes base virtuais,
- bem como nenhuma classe base com um construtor/operador/destruidor não trivial correspondente
- nenhum membro de dados de tipo de classe com um construtor/operador/destruidor não trivial correspondente

Os exemplos a seguir mostram os tipos triviais. No `Trivial2`, a presença do construtor `Trivial2(int a, int b)` requer que você forneça um construtor padrão. Para o tipo ser qualificado como trivial, você deve padronizar explicitamente o construtor.

```

struct Trivial
{
    int i;
private:
    int j;
};

struct Trivial2
{
    int i;
    Trivial2(int a, int b) : i(a), j(b) {}
    Trivial2() = default;
private:
    int j;    // Different access control
};

```

Tipos de layout padrão

Quando uma classe ou struct não contém determinados recursos da linguagem de programação C++, como funções virtuais que não são encontradas na linguagem C, e todos os membros têm o mesmo controle de acesso, essa classe ou struct é um tipo de layout padrão. Ele pode ser copiado na memória e o layout é suficientemente definido de modo que pode ser consumido por programas C. Tipos de layout padrão podem ter funções de membro especiais definidas pelo usuário. Além disso, os tipos de layout padrão têm as seguintes características:

- não têm funções virtuais nem classes base virtuais
- todos os membros de dados não estáticos têm o mesmo controle de acesso
- todos os membros não estáticos de tipo de classe são layout padrão
- as classes base são layout padrão
- não têm classes base do mesmo tipo como o primeiro membro de dados não estáticos.
- atendem a uma das seguintes condições:
 - nenhum membro de dados não estáticos na classe mais derivada e não mais de uma classe base com membros de dados não estáticos, ou
 - nenhuma classe base com membros de dados não estáticos

O código a seguir mostra um exemplo de um tipo de layout padrão:

```

struct SL
{
    // All members have same access:
    int i;
    int j;
    SL(int a, int b) : i(a), j(b) {} // User-defined constructor OK
};

```

Os dois últimos requisitos talvez possam ser mais bem ilustrados com o código. No exemplo a seguir, embora `Base` seja layout padrão, `Derived` não é layout padrão porque ambos, a classe mais derivada e `Base`, têm membros de dados não estáticos:

```

struct Base
{
    int i;
    int j;
};

// std::is_standard_layout<<Derived> == false!
struct Derived : public Base
{
    int x;
    int y;
};

```

Neste exemplo, `Derived` é layout padrão porque `Base` não tem membros de dados não estáticos:

```

struct Base
{
    void Foo() {}
};

// std::is_standard_layout<<Derived> == true
struct Derived : public Base
{
    int x;
    int y;
};

```

Derivado também seria layout padrão se `Base` tivesse membros de dados e `Derived` tivesse apenas as funções de membro.

Tipos POD

Quando uma classe ou struct é trivial e layout padrão, essa classe ou struct é um tipo POD (dados antigos simples). Portanto, o layout da memória dos tipos POD é contíguo e cada membro tem um endereço mais alto do que o membro que foi declarado antes dele para que as cópias de byte por byte e a E/S binária possam ser executadas nesses tipos. Tipos escalares, como `int`, também são tipos POD. Os tipos POD que são classes podem ter apenas tipos POD como membros de dados não estáticos.

Exemplo

O exemplo a seguir mostra as distinções entre os tipos layout padrão, trivial e POD:

```

#include <type_traits>
#include <iostream>

using namespace std;

struct B
{
protected:
    virtual void Foo() {}
};

// Neither trivial nor standard-layout
struct A : B
{
    int a;
    int b;
    void Foo() override {} // Virtual function
};

// Trivial but not standard-layout
struct C
{
    int a;
private:
    int b;    // Different access control
};

// Standard-layout but not trivial
struct D
{
    int a;
    int b;
    D() {} //User-defined constructor
};

struct POD
{
    int a;
    int b;
};

int main()
{
    cout << boolalpha;
    cout << "A is trivial is " << is_trivial<A>() << endl; // false
    cout << "A is standard-layout is " << is_standard_layout<A>() << endl; // false

    cout << "C is trivial is " << is_trivial<C>() << endl; // true
    cout << "C is standard-layout is " << is_standard_layout<C>() << endl; // false

    cout << "D is trivial is " << is_trivial<D>() << endl; // false
    cout << "D is standard-layout is " << is_standard_layout<D>() << endl; // true

    cout << "POD is trivial is " << is_trivial<POD>() << endl; // true
    cout << "POD is standard-layout is " << is_standard_layout<POD>() << endl; // true

    return 0;
}

```

Tipos literais

Um tipo literal é aquele cujo layout pode ser determinado no tempo de compilação. Os tipos a seguir são literais:

- void
- tipos escalares

- referências
- Matrizes de void, tipos escalares ou referências
- Uma classe que tem um destruidor trivial e um ou mais construtores `constexpr` que não são construtores de cópia ou movimento. Além disso, todos os seus membros de dados não estáticos e classes base devem ser tipos literais e não voláteis.

Confira também

[Conceitos Básicos](#)

Classes C++ como tipos de valor

02/12/2019 • 6 minutes to read • [Edit Online](#)

C++ as classes são por tipos de valor padrão. Eles podem ser especificados como tipos de referência, o que permite o comportamento polimórfico para dar suporte à programação orientada a objeto. Os tipos de valor às vezes são exibidos a partir da perspectiva da memória e do controle de layout, enquanto os tipos de referência são sobre classes base e funções virtuais para fins polimórficos. Por padrão, os tipos de valor são copiáveis, o que significa que há sempre um construtor de cópia e um operador de atribuição de cópia. Para tipos de referência, você torna a classe não-copiável (desabilita o construtor de cópia e o operador de atribuição de cópia) e usa um destruidor virtual que dá suporte ao polimorfismo pretendido. Os tipos de valor também são sobre o conteúdo, que, quando eles são copiados, sempre fornecem dois valores independentes que podem ser modificados separadamente. Tipos de referência são sobre identidade – que tipo de objeto é? Por esse motivo, "tipos de referência" também são chamados de "tipos polimórficos".

Se você realmente quiser um tipo de referência (classe base, funções virtuais), será necessário desabilitar explicitamente a cópia, conforme mostrado na classe `MyRefType` no código a seguir.

```
// cl /EHsc /nologo /W4

class MyRefType {
private:
    MyRefType & operator=(const MyRefType &);
    MyRefType(const MyRefType &);

public:
    MyRefType () {}
};

int main()
{
    MyRefType Data1, Data2;
    // ...
    Data1 = Data2;
}
```

A compilação do código acima resultará no seguinte erro:

```
test.cpp(15) : error C2248: 'MyRefType::operator =' : cannot access private member declared in class
'MyRefType'
        meow.cpp(5) : see declaration of 'MyRefType::operator ='
        meow.cpp(3) : see declaration of 'MyRefType'
```

Tipos de valor e eficiência de movimentação

A sobrecarga de alocação de cópia é evitada devido a novas otimizações de cópia. Por exemplo, quando você insere uma cadeia de caracteres no meio de um vetor de cadeias de caracteres, não haverá sobrecarga de realocação de cópia, apenas uma mudança, mesmo se resultar em um crescimento do próprio vetor. Isso também se aplica a outras operações, por exemplo, executando uma operação de adição em dois objetos muito grandes. Como você habilita essas otimizações de operação de valor? Em alguns C++ compiladores, o compilador permitirá isso para você implicitamente, assim como os construtores de cópia podem ser gerados automaticamente pelo compilador. No entanto C++, no, sua classe precisará "aceitar" para mover a atribuição e os construtores declarando-os em sua definição de classe. Isso é feito usando a referência a "e comercial dupla" (& &) rvalue nas declarações de função de membro apropriadas e definindo os métodos de atribuição de construtor e de movimentação de movimentação.

Você também precisa inserir o código correto para "roubar o entranhas" do objeto de origem.

Como você decide se precisa de uma movimentação habilitada? Se você já sabe que precisa de construção de cópia habilitada, provavelmente deseja mover habilitada se ela puder ser mais barata do que uma cópia em profundidade. No entanto, se você souber que precisa de suporte de movimentação, isso não significa necessariamente que você deseja que a cópia seja habilitada. Esse último caso seria chamado de "tipo somente de movimentação". Um exemplo já na biblioteca padrão é `unique_ptr`. Como uma observação adicional, o antigo `auto_ptr` é preferido e foi substituído por `unique_ptr` precisamente devido à falta de suporte à semântica de movimentação na versão anterior do C++.

Usando a semântica de movimentação, você pode retornar por valor ou inserir no meio. Move é uma otimização da cópia. Há necessidade de alocação de heap como uma solução alternativa. Considere o seguinte pseudocódigo:

```
#include <set>
#include <vector>
#include <string>
using namespace std;

//...
set<widget> LoadHugeData() {
    set<widget> ret;
    // ... load data from disk and populate ret
    return ret;
}
//...
widgets = LoadHugeData(); // efficient, no deep copy

vector<string> v = IfIHadAMillionStrings();
v.insert( begin(v)+v.size()/2, "scott" ); // efficient, no deep copy-shuffle
v.insert( begin(v)+v.size()/2, "Andrei" ); // (just 1M ptr/len assignments)
//...
HugeMatrix operator+(const HugeMatrix&, const HugeMatrix& );
HugeMatrix operator+(const HugeMatrix&, HugeMatrix&& );
HugeMatrix operator+( HugeMatrix&&, const HugeMatrix& );
HugeMatrix operator+( HugeMatrix&&, HugeMatrix&& );
//...
hm5 = hm1+hm2+hm3+hm4+hm5; // efficient, no extra copies
```

Habilitando a movimentação para tipos de valor apropriados

Para uma classe de valor em que a movimentação pode ser mais barata do que uma cópia em profundidade, habilite a construção de movimentação e a atribuição de movimento para obter eficiência. Considere o seguinte pseudocódigo:

```
#include <memory>
#include <stdexcept>
using namespace std;
// ...
class my_class {
    unique_ptr<BigHugeData> data;
public:
    my_class( my_class&& other ) // move construction
        : data( move( other.data ) ) { }
    my_class& operator=( my_class&& other ) // move assignment
    { data = move( other.data ); return *this; }
    // ...
    void method() { // check (if appropriate)
        if( !data )
            throw std::runtime_error("RUNTIME ERROR: Insufficient resources!");
    }
};
```

Se você habilitar a construção/atribuição de cópia, habilite também a construção/atribuição de movimentação se ela puder ser mais barata do que uma cópia profunda.

Alguns tipos *que não são de valor* são somente para movimentação, como quando você não pode clonar um recurso, somente transferir a propriedade. Exemplo: `unique_ptr`.

Consulte também

[C++ sistema de tipos](#)

[Bem-vindo de volta para C++](#)

[Referência da linguagem C++](#)

[Biblioteca padrão C++](#)

Conversões e segurança de tipo

02/09/2020 • 16 minutes to read • [Edit Online](#)

Este documento identifica problemas comuns de conversão de tipos e descreve como você pode evitá-los em seu código C++.

Ao escrever um programa em C++, é importante garantir que ele seja de tipo seguro. Isso significa que cada variável, um argumento de função e um valor de retorno de função estão armazenando um tipo aceitável de dados e as operações que envolvem valores de diferentes tipos "fazem sentido" e não causam perda de dados, interpretação incorreta de padrões de bits ou corrupção de memória. Um programa que nunca converte valores explicitamente ou implicitamente de um tipo para outro é de tipo seguro por definição. No entanto, as conversões de tipo, mesmo as conversões não seguras, às vezes são necessárias. Por exemplo, talvez você precise armazenar o resultado de uma operação de ponto flutuante em uma variável de tipo `int`, ou pode ser necessário passar o valor em um `unsigned int` para uma função que usa um `signed int`. Ambos os exemplos ilustram conversões não seguras porque podem causar perda de dados ou nova interpretação de um valor.

Quando o compilador detecta uma conversão não segura, ele emite um erro ou um aviso. Um erro para a compilação; um aviso permite que a compilação Continue, mas indica um possível erro no código. No entanto, mesmo que o programa seja compilado sem avisos, ele ainda pode conter código que leva a conversões implícitas de tipo que produzem resultados incorretos. Os erros de tipo também podem ser introduzidos por conversões explícitas, ou conversões no código.

Conversões de tipo implícitas

Quando uma expressão contém operandos de diferentes tipos internos e nenhuma conversão explícita está presente, o compilador usa *conversões padrão* internas para converter um dos operandos para que os tipos sejam correspondentes. O compilador tenta as conversões em uma sequência bem definida até obter um sucesso. Se a conversão selecionada for uma promoção, o compilador não emitirá um aviso. Se a conversão for um estreitamento, o compilador emitirá um aviso sobre possível perda de dados. A perda de dados real depende dos valores reais envolvidos, mas recomendamos que você trate esse aviso como um erro. Se um tipo definido pelo usuário estiver envolvido, o compilador tentará usar as conversões que você especificou na definição de classe. Se não for possível encontrar uma conversão aceitável, o compilador emitirá um erro e não compilará o programa. Para obter mais informações sobre as regras que regem as conversões padrão, consulte [conversões padrão](#). Para obter mais informações sobre conversões definidas pelo usuário, consulte [conversões definidas pelo usuário \(C++/CLI\)](#).

Conversões de alargamento (promoção)

Em uma conversão de ampliação, um valor em uma variável menor é atribuído a uma variável maior sem perda de dados. Como as conversões de alargamento são sempre seguras, o compilador as executa silenciosamente e não emite avisos. As conversões a seguir são conversões ampliadas.

DE	PARA
Qualquer <code>signed</code> <code>unsigned</code> tipo ou integral, exceto <code>long long</code> ou** <code>__int64</code> **	<code>double</code>
<code>bool</code> or** <code>char</code>	Qualquer outro tipo interno
<code>short</code> or** <code>wchar_t</code>	<code>int</code> , <code>long</code> , <code>long long</code>

DE	PARA
<code>int</code> , <code>long</code>	<code>long long</code>
<code>float</code>	<code>double</code>

Refinando conversões (coerção)

O compilador executa conversões estreitas implicitamente, mas avisa sobre potencial perda de dados. Considere esses avisos muito seriamente. Se você tiver certeza de que nenhuma perda de dados ocorrerá porque os valores na variável maior sempre se ajustarão à variável menor e, em seguida, adicionará uma conversão explícita para que o compilador não emita mais um aviso. Se você não tiver certeza de que a conversão é segura, adicione ao seu código algum tipo de verificação de tempo de execução para lidar com a possível perda de dados para que ele não faça com que o programa produza resultados incorretos.

Qualquer conversão de um tipo de ponto flutuante para um tipo integral é uma conversão de restrição, pois a parte fracionária do valor de ponto flutuante é descartada e perdida.

O exemplo de código a seguir mostra algumas conversões de estreitamento implícitas e os avisos que o compilador emite para elas.

```
int i = INT_MAX + 1; //warning C4307:'+' : integral constant overflow
wchar_t wch = 'A'; //OK
char c = wch; // warning C4244: 'initializing' : conversion from 'wchar_t'
               // to 'char', possible loss of data
unsigned char c2 = 0xfffe; //warning C4305: 'initializing' : truncation from
                           // 'int' to 'unsigned char'
int j = 1.9f; // warning C4244: 'initializing' : conversion from 'float' to
               // 'int', possible loss of data
int k = 7.7; // warning C4244: 'initializing' : conversion from 'double' to
               // 'int', possible loss of data
```

Conversões assinadas com sinal

Um tipo integral assinado e sua contraparte não assinada são sempre do mesmo tamanho, mas diferem em como o padrão de bit é interpretado para transformação de valor. O exemplo de código a seguir demonstra o que acontece quando o mesmo padrão de bit é interpretado como um valor assinado e como um valor não assinado. O padrão de bit armazenado em `num` e `num2` nunca altera o que é mostrado na ilustração anterior.

```
using namespace std;
unsigned short num = numeric_limits<unsigned short>::max(); // #include <limits>
short num2 = num;
cout << "unsigned val = " << num << " signed val = " << num2 << endl;
// Prints: unsigned val = 65535 signed val = -1

// Go the other way.
num2 = -1;
num = num2;
cout << "unsigned val = " << num << " signed val = " << num2 << endl;
// Prints: unsigned val = 65535 signed val = -1
```

Observe que os valores são reinterpretados em ambas as direções. Se o seu programa produzir resultados estranhos em que o sinal do valor pareça invertido do esperado, procure conversões implícitas entre tipos integrais assinados e não assinados. No exemplo a seguir, o resultado da expressão (0-1) é implicitamente convertido de `int` para `unsigned int` quando ele é armazenado em `num`. Isso faz com que o padrão de bit seja reinterpretado.

```
unsigned int u3 = 0 - 1;
cout << u3 << endl; // prints 4294967295
```

O compilador não avisa sobre conversões implícitas entre tipos integrais assinados e não assinados. Portanto, recomendamos que você evite totalmente as conversões assinadas sem assinatura. Se você não puder evitá-los, adicione ao seu código uma verificação de tempo de execução para detectar se o valor que está sendo convertido é maior ou igual a zero e menor ou igual ao valor máximo do tipo assinado. Os valores neste intervalo serão transferidos de assinados para não assinados ou de não assinados para assinados sem serem reinterpretados.

Conversões de ponteiro

Em muitas expressões, uma matriz C-Style é implicitamente convertida em um ponteiro para o primeiro elemento na matriz, e conversões constantes podem ocorrer silenciosamente. Embora isso seja conveniente, ele também é potencialmente propenso a erros. Por exemplo, o seguinte exemplo de código criado incorretamente parece não-e, ainda assim, será compilado e produzirá um resultado de "p". Primeiro, o literal de constante de cadeia de caracteres "Help" é convertido em um `char*` que aponta para o primeiro elemento da matriz; esse ponteiro é incrementado em três elementos para que agora aponte para o último elemento "p".

```
char* s = "Help" + 3;
```

Conversões explícitas (difusões)

Usando uma operação de conversão, você pode instruir o compilador a converter um valor de um tipo para outro tipo. O compilador gerará um erro em alguns casos se os dois tipos estiverem completamente não relacionados, mas em outros casos ele não gerará um erro mesmo que a operação não seja de tipo seguro. Use as conversões com moderação porque qualquer conversão de um tipo para outro é uma fonte potencial de erro do programa. No entanto, às vezes as conversões são necessárias, e nem todas as conversões são igualmente perigosas. Um uso eficaz de uma conversão é quando seu código executa uma conversão de restrição e você sabe que a conversão não está fazendo com que seu programa produza resultados incorretos. Na verdade, isso informa ao compilador que você sabe o que está fazendo e para parar de incomodar com avisos sobre ele. Outro uso é converter de uma classe de ponteiro para derivada para uma classe de ponteiro para base. Outro uso é converter o `const` -qualidade de uma variável para passá-la para uma função que requer um não `const` argumento. A maioria dessas operações de conversão envolve alguns riscos.

Na programação em estilo C, o mesmo operador de conversão C-Style é usado para todos os tipos de conversões.

```
(int) x; // old-style cast, old-style syntax
int(x); // old-style cast, functional syntax
```

O operador de conversão C-Style é idêntico ao operador de chamada () e, portanto, é inevidente no código e fácil de ignorar. Ambos são ruins porque são difíceis de reconhecer rapidamente ou Pesquisar, e são diferentes o suficiente para invocar qualquer combinação de `static`, `const` e `reinterpret_cast`. Descobrir o que uma conversão de estilo antigo realmente pode ser difícil e propenso a erros. Por todos esses motivos, quando uma conversão é necessária, recomendamos que você use um dos seguintes operadores de conversão de C++, que, em alguns casos, são significativamente mais fortemente tipados e que expressam muito mais explicitamente a intenção de programação:

- `static_cast`, para conversões que são verificadas somente no momento da compilação. `static_cast` retornará um erro se o compilador detectar que você está tentando converter entre tipos que são completamente incompatíveis. Você também pode usá-lo para converter entre ponteiro para base e ponteiro para derivado, mas o compilador nem sempre informa se essas conversões serão seguras no

tempo de execução.

```
double d = 1.58947;
int i = d; // warning C4244 possible loss of data
int j = static_cast<int>(d); // No warning.
string s = static_cast<string>(d); // Error C2440:cannot convert from
// double to std::string

// No error but not necessarily safe.
Base* b = new Base();
Derived* d2 = static_cast<Derived*>(b);
```

Para obter mais informações, consulte [static_cast](#).

- `dynamic_cast`, para conversões seguras e verificadas em tempo de execução de ponteiro para base a ponteiro para derivado. R `dynamic_cast` é mais seguro do que um `static_cast` for downcasts, mas a verificação de tempo de execução incorre em sobrecarga.

```
Base* b = new Base();

// Run-time check to determine whether b is actually a Derived*
Derived* d3 = dynamic_cast<Derived*>(b);

// If b was originally a Derived*, then d3 is a valid pointer.
if(d3)
{
    // Safe to call Derived method.
    cout << d3->DoSomethingMore() << endl;
}
else
{
    // Run-time check failed.
    cout << "d3 is null" << endl;
}

//Output: d3 is null;
```

Para obter mais informações, consulte [dynamic_cast](#).

- `const_cast`, para a conversão de `const` -qualidade de uma variável ou a conversão de uma não `const` variável como `const`. A transmissão `const` de qualidade usando esse operador é tão propenso a erros quanto o uso de uma conversão C-Style, exceto que com `const_cast` você tem menos probabilidade de executar a conversão acidentalmente. Às vezes, você precisa converter o `const` -qualidade de uma variável, por exemplo, para passar uma `const` variável para uma função que usa um não `const` parâmetro. O exemplo a seguir mostra como fazer isso.

```
void Func(double& d) { ... }
void ConstCast()
{
    const double pi = 3.14;
    Func(const_cast<double&>(pi)); //No error.
}
```

Para obter mais informações, consulte [const_cast](#).

- `reinterpret_cast`, para conversões entre tipos não relacionados, como um tipo de ponteiro e um `int`.

NOTE

Esse operador cast não é usado com frequência como os outros, e não é garantido que seja portátil para outros compiladores.

O exemplo a seguir ilustra como o `reinterpret_cast` difere do `static_cast`.

```
const char* str = "hello";
int i = static_cast<int>(str); //error C2440: 'static_cast' : cannot
                                // convert from 'const char **' to 'int'
int j = (int)str; // C-style cast. Did the programmer really intend
                  // to do this?
int k = reinterpret_cast<int>(str); // Programming intent is clear.
                                    // However, it is not 64-bit safe.
```

Para obter mais informações, consulte [operator `reinterpret_cast`](#).

Confira também

[Sistema de tipos C++](#)

[Bem-vindo de volta ao C++](#)

[Referência da linguagem C++](#)

[Biblioteca padrão do C++](#)

Conversões padrão

02/09/2020 • 24 minutes to read • [Edit Online](#)

A linguagem C++ define conversões entre seus tipos básicos. Ela também define conversões para o ponteiro, referência e tipos derivados de ponteiro ao membro. Essas conversões são chamadas de *conversões padrão*.

Esta seção aborda as seguintes conversões padrão:

- Promoções de integral
- Conversões de integral
- Conversões flutuantes
- Conversões flutuantes e integrais
- Conversões aritméticas
- Conversões de ponteiro
- Conversões de referência
- Conversões de ponteiro ao membro

NOTE

Os tipos definidos pelo usuário podem especificar suas próprias conversões. A conversão de tipos definidos pelo usuário é abordada em [construtores e conversões](#).

O código a seguir causa conversões (neste exemplo, promoções de integral):

```
long long_num1, long_num2;
int int_num;

// int_num promoted to type long prior to assignment.
long_num1 = int_num;

// int_num promoted to type long prior to multiplication.
long_num2 = int_num * long_num2;
```

O resultado de uma conversão é um l-value somente se ela produzir um tipo de referência. Por exemplo, uma conversão definida pelo usuário declarada como `operator int&()` retorna uma referência e é um l-Value. No entanto, uma conversão declarada como `operator int()` retorna um objeto e não é um l-Value.

Promoções de integral

Objetos de um tipo integral podem ser convertidos em outro tipo integral mais amplo, ou seja, um tipo que pode representar um conjunto maior de valores. Esse tipo de ampliação de conversão é chamado de *promoção integral*. Com a promoção integral, você pode usar os seguintes tipos em uma expressão onde quer que outro tipo integral possa ser usado:

- Objetos, literais e constantes do tipo `char` `e**` `short` `int` `**`
- Tipos de enumeração

- `int` campos de bits

- Enumeradores

As promoções de C++ são "de preservação de valor", pois o valor após a promoção tem a garantia de ser igual ao valor antes da promoção. Nas promoções de preservação de valor, os objetos de tipos de integral mais curtos (como campos de bits ou objetos do tipo `char`) são promovidos para `int` o tipo se `int` o puder representar o intervalo completo do tipo original. Se `int` não puder representar o intervalo completo de valores, o objeto será promovido para Type `unsigned int`. Embora essa estratégia seja a mesma usada pelas conversões padrão C, a preservação de valores não preserva a "assinatura" do objeto.

As promoções de preservação de valores e as promoções que preservam o signedness normalmente geram os mesmos resultados. No entanto, eles podem produzir resultados diferentes se o objeto promovido aparecer como:

- Um operando do `"/`, `%`, `/=`, `%=`, `<`, `<=`, `>` ou `>=`

Esses operadores dependem do sinal para determinar o resultado. A preservação de valor e a preservação de promoções geram resultados diferentes quando aplicadas a esses operandos.

- O operando esquerdo de `>>` ou `>>=`

Esses operadores tratam quantidades assinadas e não assinadas de forma diferente em uma operação de deslocamento. Para quantidades assinadas, uma operação de deslocamento à direita propaga o bit de sinal para as posições de bits vagadas, enquanto as posições de bits vagadas são preenchidas com zero em quantidades não assinadas.

- Um argumento para uma função sobrecarregada ou o operando de um operador sobrecarregado, que depende da assinatura do tipo de operando para correspondência de argumento. Para obter mais informações sobre como definir operadores sobrecarregados, consulte [operadores sobrecarregados](#).

Conversões de integral

Conversões integrais são conversões entre tipos integrais. Os tipos integrais são `char`, `short` (ou `short int`), `int`, `long` e `long long`. Esses tipos podem ser qualificados com `signed` ou `unsigned`, e `unsigned` podem ser usados como abreviação para `unsigned int`.

Assinado para não assinado

Os objetos de tipos integrais com sinal podem ser convertidos nos tipos sem sinal correspondentes. Quando essas conversões ocorrem, o padrão de bit real não é alterado. No entanto, a interpretação dos dados é alterada. Considere este código:

```
#include <iostream>

using namespace std;
int main()
{
    short i = -3;
    unsigned short u;

    cout << (u = i) << "\n";
}
```

// Output: 65533

No exemplo anterior, um `signed short`, `i`, é definido e inicializado com um número negativo. A expressão `(u = i)` faz com que `i` seja convertida em um `unsigned short` antes da atribuição para `u`.

Não assinado para assinado

Os objetos de tipos integrais sem sinal podem ser convertidos nos tipos com sinal correspondentes. No entanto, se o valor não assinado estiver fora do intervalo representável do tipo assinado, o resultado não terá o valor correto, conforme demonstrado no exemplo a seguir:

```
#include <iostream>

using namespace std;
int main()
{
short i;
unsigned short u = 65533;

cout << (i = u) << "\n";
}
//Output: -3
```

No exemplo anterior, `u` é um `unsigned short` objeto integral que deve ser convertido em uma quantidade assinada para avaliar a expressão `(i = u)`. Como seu valor não pode ser adequadamente representado em um `signed short`, os dados são interpretados incorretamente, conforme mostrado.

Conversões de ponto flutuante

Um objeto de um tipo flutuante pode ser convertido com segurança em um tipo flutuante mais preciso; ou seja, a conversão não resulta em nenhuma perda de significação. Por exemplo, conversões de `float` para `double` ou de `double` para `long double` são seguras e o valor não é alterado.

Um objeto de um tipo flutuante também pode ser convertido para um tipo menos preciso, se ele estiver em um intervalo representável por esse tipo. (Consulte [limites flutuantes](#) para os intervalos de tipos flutuantes.) Se o valor original não for rerepresentável precisamente, ele poderá ser convertido para o próximo valor mais alto ou mais baixo representável. O resultado será indefinido se nenhum valor desse tipo existir. Considere o exemplo a seguir:

```
cout << (float)1E300 << endl;
```

O valor máximo representável por tipo `float` é 3.402823466 E38 – um número muito menor que 1E300. Portanto, o número é convertido em infinito e o resultado é "inf".

Conversões entre tipos integral e de ponto flutuante

Algumas expressões podem fazer com que os objetos de tipo flutuante sejam convertidos em tipos integrais, ou vice-versa. Quando um objeto do tipo integral é convertido em um tipo flutuante, e o valor original não é rerepresentável exatamente, o resultado é o próximo valor mais alto ou mais baixo representável.

Quando um objeto de tipo flutuante é convertido em um tipo integral, a parte fracionária é *truncada* ou arredondada para zero. Um número como 1,3 é convertido em 1 e -1,3 é convertido em -1. Se o valor truncado for maior do que o maior valor representável, ou menor que o menor valor representável, o resultado será indefinido.

Conversões aritméticas

Muitos operadores binários (discutidos em [expressões com operadores binários](#)) causam conversões de operandos e geram resultados da mesma maneira. As conversões desses operadores causam chamadas de *conversões aritméticas usuais*. Conversões aritméticas de operandos que têm tipos nativos diferentes são feitas, conforme mostrado na tabela a seguir. Os tipos `typedef` se comportam de acordo com seus tipos nativos subjacentes.

Condições para conversão de tipo

CONDIÇÕES ATENDIDAS	CONVERSÃO
Qualquer operando é do tipo <code>long double</code> .	Outro operando é convertido para o tipo <code>long double</code> .
A condição anterior não foi atendida e o operando é do tipo <code>double</code> .	Outro operando é convertido para o tipo <code>double</code> .
As condições anteriores não foram atendidas e o operando é do tipo <code>float</code> .	Outro operando é convertido para o tipo <code>float</code> .
As condições anteriores não foram atendidas (nenhum dos operandos é de tipo flutuante).	<p>Os operandos obtêm promoções integrais da seguinte maneira:</p> <ul style="list-style-type: none"> -Se qualquer operando for do tipo <code>unsigned long</code> , o outro operando será convertido em tipo <code>unsigned long</code> . -Se a condição anterior não for atendida e se um dos operandos for do tipo <code>long</code> e o outro do tipo <code>unsigned int</code> , ambos os operandos serão convertidos para o tipo <code>unsigned long</code> . -Se as duas condições anteriores não forem atendidas e se um dos operandos for do tipo <code>long</code> , o outro operando será convertido em tipo <code>long</code> . -Se as três condições anteriores não forem atendidas e se um dos operandos for do tipo <code>unsigned int</code> , o outro operando será convertido em tipo <code>unsigned int</code> . -Se nenhuma das condições anteriores for atendida, ambos os operandos serão convertidos para o tipo <code>int</code> .

O código a seguir ilustra as regras de conversão descritas na tabela:

```
double dVal;
float fVal;
int iVal;
unsigned long ulVal;

int main() {
    // iVal converted to unsigned long
    // result of multiplication converted to double
    dVal = iVal * ulVal;

    // ulVal converted to float
    // result of addition converted to double
    dVal = ulVal + fVal;
}
```

A primeira instrução no exemplo acima mostra a multiplicação de dois tipos integrais, `iVal` e `ulVal` . A condição atendida é que nenhum operando é do tipo flutuante e um operando é do tipo `unsigned int` . Portanto, o outro operando, `iVal` , é convertido em tipo `unsigned int` . O resultado é então atribuído a `dVal` . A condição atendida aqui é que um operando é do tipo `double` , portanto, o `unsigned int` resultado da multiplicação é convertido em tipo `double` .

A segunda instrução no exemplo anterior mostra a adição de um `float` e um tipo integral: `fVal` e `ulVal` . A `ulVal` variável é convertida no tipo `float` (terceira condição na tabela). O resultado da adição é convertido para o tipo `double` (segunda condição na tabela) e atribuído a `dVal` .

Conversões de ponteiro

Os ponteiros podem ser convertidos durante a atribuição, a inicialização, a comparação e outras expressões.

Ponteiro para classes

Há dois casos nos quais um ponteiro para uma classe pode ser convertido em um ponteiro para uma classe base.

O primeiro é quando a classe base especificada estiver acessível e a conversão for inequívoca. Para obter mais informações sobre referências de classe base ambíguas, consulte [várias classes base](#).

O acesso a uma classe base depende do tipo de herança usado na derivação. Considere a herança ilustrada na figura a seguir.

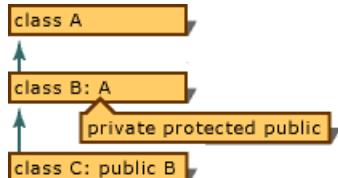


Gráfico de herança para ilustração de acessibilidade a classe base

A tabela a seguir mostra a acessibilidade da classe base para a situação ilustrada na figura.

TIPO DE FUNÇÃO	DERIVAÇÃO	CONVERSÃO DE B * PARA UMA PESSOA * JURÍDICA?
Função externa (fora do escopo da classe)	Privado	Não
	Protegido	Não
	Público	Sim
Função membro B (no escopo de B)	Privado	Sim
	Protegido	Sim
	Público	Sim
Função membro C (no escopo de C)	Privado	Não
	Protegido	Sim
	Público	Sim

O segundo caso em que um ponteiro para uma classe pode ser convertido em um ponteiro para uma classe base é quando uma conversão de tipo explícita. Para obter mais informações sobre conversões de tipo explícitas, consulte [operador de conversão de tipo explícito](#).

O resultado de tal conversão é um ponteiro para o *subobjeto*, a parte do objeto que é completamente descrito pela classe base.

O código a seguir define duas classes, `A` e `B`, onde `B` é derivado de `A`. (Para obter mais informações sobre herança, consulte [classes derivadas](#).) Em seguida, ele define `bObject`, um objeto do tipo `B` e dois ponteiros (`pA` e `pB`) que apontam para o objeto.

```

// C2039 expected
class A
{
public:
    int AComponent;
    int AMemberFunc();
};

class B : public A
{
public:
    int BComponent;
    int BMemberFunc();
};

int main()
{
    B bObject;
    A *pA = &bObject;
    B *pB = &bObject;

    pA->AMemberFunc(); // OK in class A
    pB->AMemberFunc(); // OK: inherited from class A
    pA->BMemberFunc(); // Error: not in class A
}

```

O ponteiro `pA` é do tipo `A *`, o que pode ser interpretado como "ponteiro para um objeto do tipo `A`". Os membros de `bObject` (como `BComponent` e `BMemberFunc`) são exclusivos do tipo `B` e, portanto, são inacessíveis por meio do `pA`. O ponteiro `pA` permite acesso somente às características (funções membro e dados) do objeto que são definidas na classe `A`.

Ponteiro para função

Um ponteiro para uma função pode ser convertido em tipo `void *`, se o tipo `void *` for grande o suficiente para manter esse ponteiro.

Ponteiro para void

Ponteiros para tipo `void` podem ser convertidos em ponteiros para qualquer outro tipo, mas apenas com uma conversão explícita de tipo (ao contrário de C). Um ponteiro para qualquer tipo pode ser convertido implicitamente em um ponteiro para tipo `void`. Um ponteiro para um objeto incompleto de um tipo pode ser convertido em um ponteiro para `void` (implicitamente) e voltar (explicitamente). O resultado dessa conversão é igual ao valor do ponteiro original. Um objeto é considerado incompleto se for declarado, mas não há informações suficientes disponíveis para determinar seu tamanho ou classe base.

Um ponteiro para qualquer objeto que não seja `const` ou `volatile` pode ser convertido implicitamente em um ponteiro do tipo `void *`.

Ponteiros `const` e `volatile`

O C++ não fornece uma conversão padrão de `const` um `volatile` tipo ou para um tipo que não seja `const` ou `volatile`. No entanto, qualquer tipo de conversão pode ser especificado usando as conversões de tipos explícitas (inclusive as conversões que não são seguras).

NOTE

Os ponteiros do C++ para membros, exceto ponteiros para membros estáticos, são diferentes dos ponteiros normais e não têm as mesmas conversões padrão. Os ponteiros para os membros estáticos são ponteiros normais e têm as mesmas conversões que ponteiros normais.

conversões de ponteiro NULL

Uma expressão constante integral que é avaliada como zero, ou tal conversão de expressão em um tipo de ponteiro, é convertida em um ponteiro chamado de *ponteiro NULL*. Esse ponteiro sempre compara desigual a um ponteiro para qualquer objeto ou função válida. Uma exceção é ponteiros para objetos baseados, que podem ter o mesmo deslocamento e ainda apontar para objetos diferentes.

No C++ 11, o tipo `nullptr` deve ser preferível ao ponteiro NULL de estilo C.

Conversões de expressão de ponteiro

Qualquer expressão com um tipo de matriz pode ser convertida em um ponteiro do mesmo tipo. O resultado da conversão é um ponteiro para o primeiro elemento da matriz. O exemplo a seguir demonstra essa conversão:

```
char szPath[_MAX_PATH]; // Array of type char.  
char *pszPath = szPath; // Equals &szPath[0].
```

Uma expressão que resulta em uma função que retorna um tipo específico é convertido em um ponteiro para uma função que retorna esse tipo, exceto quando:

- A expressão é usada como um operando para o operador address-of (`&`).
- A expressão é usada como um operando para o operador function-call.

Conversões de referência

Uma referência a uma classe pode ser convertida em uma referência a uma classe base nesses casos:

- A classe base especificada está acessível.
- A conversão é inequívoca. (Para obter mais informações sobre referências de classe base ambíguas, consulte [várias classes base](#).)

O resultado da conversão é um ponteiro para o subobjeto que representa a classe base.

Ponteiro para membro

Os ponteiros para membros de classe podem ser convertidos durante a atribuição, a inicialização, a comparação e outras expressões. Esta seção descreve as seguintes conversões de ponteiros para membros:

Ponteiro para membro da classe base

Um ponteiro para um membro de uma classe base pode ser convertido em um ponteiro para um membro de uma classe derivada dela, quando as seguintes condições são atendidas:

- A conversão inversa, de ponteiro para classe derivada em ponteiro para classe base, é acessível.
- Não há herança virtual entre a classe base e a classe derivada.

Quando o operando esquerdo é um ponteiro para um membro, o operando direito deve ser do tipo ponteiro para membro ou ser uma expressão de constante que é avaliada como 0. Essa atribuição só é válida nos seguintes casos:

- O operando direito é um ponteiro para um membro da mesma classe que o operando esquerdo.
- O operando esquerdo é um ponteiro para um membro de uma classe derivada, de forma pública e sem ambiguidade, da classe do operando direito.

ponteiro nulo para conversões de membro

Uma expressão constante integral que é avaliada como zero é convertida em um ponteiro nulo. Esse ponteiro sempre compara desigual a um ponteiro para qualquer objeto ou função válida. Uma exceção é ponteiros

para objetos baseados, que podem ter o mesmo deslocamento e ainda apontar para objetos diferentes.

O código a seguir ilustra a definição de um ponteiro para o membro `i` na classe `A`. O ponteiro, `pai`, é inicializado como 0, que é o ponteiro nulo.

```
class A
{
public:
    int i;
};

int A::*pai = 0;

int main()
{
```

Confira também

[Referência da linguagem C++](#)

Tipos internos (C++)

02/09/2020 • 11 minutes to read • [Edit Online](#)

Os tipos internos (também chamados de *tipos fundamentais*) são especificados pelo padrão de linguagem C++ e são criados no compilador. Os tipos internos não são definidos em nenhum arquivo de cabeçalho. Os tipos internos são divididos em três categorias principais: *integral*, *ponto flutuante* e *void*. Tipos integrais representam números inteiros. Tipos de ponto flutuante podem especificar valores que podem ter partes fracionárias. A maioria dos tipos internos são tratados como tipos distintos pelo compilador. No entanto, alguns tipos são *sinônimos* ou tratados como tipos equivalentes pelo compilador.

Tipo void

O `void` tipo descreve um conjunto vazio de valores. Nenhuma variável do tipo `void` pode ser especificada. O `void` tipo é usado principalmente para declarar funções que não retornam valores ou para declarar ponteiros genéricos para dados digitados de forma arbitrária ou sem tipo. Qualquer expressão pode ser convertida explicitamente ou converter em tipo `void`. No entanto, tais expressões estão restritas aos seguintes usos:

- Uma instrução de expressão. (Para obter mais informações, consulte [Expressions](#).)
- O operando esquerdo do operador vírgula. (Para obter mais informações, consulte [operador de vírgula](#).)
- O segundo ou terceiro operando do operador condicional (`? :`). (Para obter mais informações, consulte [expressões com o operador condicional](#).)

std::nullptr_t

A palavra-chave `nullptr` é uma constante de ponteiro nulo do tipo `std::nullptr_t`, que é conversível para qualquer tipo de ponteiro bruto. Para obter mais informações, consulte [nullptr](#).

Tipo booleano

O `bool` tipo pode ter valores `true` e `false`. O tamanho do `bool` tipo é específico da implementação. Consulte [tamanhos de tipos internos](#) para obter detalhes de implementação específicos da Microsoft.

Tipos de caractere

O `char` tipo é um tipo de representação de caractere que codifica com eficiência os membros do conjunto de caracteres de execução básico. O compilador C++ trata variáveis do tipo `char`, `signed char` e `unsigned char` como tendo tipos diferentes.

Específico da Microsoft: variáveis do tipo `char` são promovidas para `int` como If do tipo `signed char` por padrão, a menos que a `/J` opção de compilação seja usada. Nesse caso, eles são tratados como tipo `unsigned char` e são promovidos para `int` sem extensão de assinatura.

Uma variável do tipo `wchar_t` é um tipo de caractere largo ou multibyte. Use o `L` prefixo antes de um literal de caractere ou de cadeia de caracteres para especificar o tipo de caractere largo.

Específico da Microsoft: por padrão, `wchar_t` é um tipo nativo, mas você pode usar `/Zc:wchar_t-` para criar `wchar_t` um typedef para `unsigned short`. O `_wchar_t` tipo é um sinônimo específico da Microsoft

para o `wchar_t` tipo nativo.

O `char8_t` tipo é usado para a representação de caracteres UTF-8. Ele tem a mesma representação que `unsigned char`, mas é tratado como um tipo distinto pelo compilador. O `char8_t` tipo é novo no C++ 20. **Específico da Microsoft:** o uso de `char8_t` requer a `/std:c++latest` opção de compilador.

O `char16_t` tipo é usado para a representação de caracteres UTF-16. Ele deve ser grande o suficiente para representar qualquer unidade de código UTF-16. Ele é tratado como um tipo distinto pelo compilador.

O `char32_t` tipo é usado para a representação de caractere UTF-32. Ele deve ser grande o suficiente para representar qualquer unidade de código UTF-32. Ele é tratado como um tipo distinto pelo compilador.

Tipos de ponto flutuante

Os tipos de ponto flutuante usam uma representação IEEE-754 para fornecer uma aproximação de valores fracionários em uma ampla gama de magnitudes. A tabela a seguir lista os tipos de ponto flutuante em C++ e as restrições comparativa em tamanhos de tipo de ponto flutuante. Essas restrições são obrigatórias pelo padrão C++ e são independentes da implementação da Microsoft. O tamanho absoluto dos tipos de ponto flutuante internos não é especificado no padrão.

TYPE	SUMÁRIO
<code>float</code>	Type <code>float</code> é o menor tipo de ponto flutuante em C++.
<code>double</code>	Type <code>double</code> é um tipo de ponto flutuante que é maior ou igual ao tipo <code>float</code> , mas menor que ou igual ao tamanho do tipo <code>long double</code> .
<code>long double</code>	Type <code>long double</code> é um tipo de ponto flutuante que é maior ou igual ao tipo <code>double</code> .

Específico da Microsoft: a representação de `long double` e `double` é idêntica. No entanto, `long double` e `double` são tratados como tipos distintos pelo compilador. O compilador do Microsoft C++ usa as representações de ponto flutuante de 4 e 8 bytes IEEE-754. Para obter mais informações, consulte [representação de ponto flutuante IEEE](#).

Tipos de inteiro

O `int` tipo é o tipo de inteiro básico padrão. Ele pode representar todos os números inteiros em um intervalo específico de implementação.

Uma representação de inteiro *assinada* é aquela que pode conter valores positivos e negativos. Ele é usado por padrão ou quando a `signed` palavra-chave do modificador está presente. A `unsigned` palavra-chave modificador especifica uma representação não *assinada* que só pode conter valores não negativos.

Um modificador de tamanho especifica a largura em bits da representação de inteiro usada. O idioma dá suporte a `short` `long` `long long` modificadores, e. Um `short` tipo deve ter pelo menos 16 bits de largura. Um `long` tipo deve ter pelo menos 32 bits de largura. Um `long long` tipo deve ter pelo menos 64 bits de largura. O padrão especifica uma relação de tamanho entre os tipos inteiros:

```
1 == sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long) <= sizeof(long long)
```

Uma implementação deve manter os requisitos de tamanho mínimo e a relação de tamanho para cada tipo. No entanto, os tamanhos reais podem variar entre as implementações. Consulte [tamanhos de tipos internos](#) para obter detalhes de implementação específicos da Microsoft.

A `int` palavra-chave pode ser omitida quando os `signed` `unsigned` modificadores de tamanho, ou são especificados. Os modificadores e o `int` tipo, se presentes, podem aparecer em qualquer ordem. Por exemplo, `short unsigned` e `unsigned int short` se referem ao mesmo tipo.

Sinônimos de tipo Integer

Os seguintes grupos de tipos são considerados sinônimos pelo compilador:

- `short`, `short int`, `signed short`, `signed short int`
- `unsigned short`, `unsigned short int`
- `int`, `signed`, `signed int`
- `unsigned`, `unsigned int`
- `long`, `long int`, `signed long`, `signed long int`
- `unsigned long`, `unsigned long int`
- `long long`, `long long int`, `signed long long`, `signed long long int`
- `unsigned long long`, `unsigned long long int`

Os tipos inteiros **específicos da Microsoft** incluem os tipos de largura específica `_int8`, `_int16`, `_int32` e `_int64`. Esses tipos podem usar os `signed` `unsigned` modificadores e. O `_int8` tipo de dados é sinônimo de tipo `char`, `_int16` é sinônimo de tipo `short`, `_int32` é sinônimo de tipo `int` e `_int64` é sinônimo de tipo `long long`.

Tamanhos de tipos internos

A maioria dos tipos internos tem tamanhos definidos pela implementação. A tabela a seguir lista a quantidade de armazenamento necessária para tipos internos no Microsoft C++. Em particular, o `long` é de 4 bytes até mesmo em sistemas operacionais de 64 bits.

TYPE	TAMANHO
<code>bool</code> , <code>char</code> , <code>char8_t</code> , <code>unsigned char</code> , <code>signed char</code> , <code>_int8</code>	1 byte
<code>char16_t</code> , <code>_int16</code> , <code>short</code> , <code>unsigned short</code> , <code>wchar_t</code> , <code>_wchar_t</code>	2 bytes
<code>char32_t</code> , <code>float</code> , <code>_int32</code> , <code>int</code> , <code>unsigned int</code> , <code>long</code> , <code>unsigned long</code>	4 bytes
<code>double</code> , <code>_int64</code> , <code>long double</code> , <code>long long</code> , <code>unsigned long long</code>	8 bytes

Consulte [intervalos de tipos de dados](#) para obter um resumo do intervalo de valores de cada tipo.

Para obter mais informações sobre conversão de tipo, consulte [conversões padrão](#).

Confira também

[Intervalos de tipos de dados](#)

Intervalos de tipos de dados

02/09/2020 • 4 minutes to read • [Edit Online](#)

Os compiladores do Microsoft C++ 32 bits e 64 bits reconhecem os tipos na tabela mais adiante neste artigo.

- `int` (`unsigned int`)
- `__int8` (`unsigned __int8`)
- `__int16` (`unsigned __int16`)
- `__int32` (`unsigned __int32`)
- `__int64` (`unsigned __int64`)
- `short` (`unsigned short`)
- `long` (`unsigned long`)
- `long long` (`unsigned long long`)

Se o nome começa com sublinhados duplos (`__`), um tipo de dados é diferente do padrão.

Os intervalos especificados na tabela a seguir são inclusivo-inclusivo.

NOME DO TIPO	BYTES	OUTROS NOMES	INTERVALO DE VALORES
<code>int</code>	4	<code>signed</code>	-2.147.483.648 a 2.147.483.647
<code>unsigned int</code>	4	<code>unsigned</code>	0 a 4.294.967.295
<code>__int8</code>	1	<code>char</code>	-128 a 127
<code>unsigned __int8</code>	1	<code>unsigned char</code>	0 a 255
<code>__int16</code>	2	<code>short</code> , <code>short int</code> , <code>signed short int</code>	-32.768 a 32.767
<code>unsigned __int16</code>	2	<code>unsigned short</code> , <code>unsigned short int</code>	0 a 65.535
<code>__int32</code>	4	<code>signed</code> , <code>signed int</code> , <code>int</code>	-2.147.483.648 a 2.147.483.647
<code>unsigned __int32</code>	4	<code>unsigned</code> , <code>unsigned int</code>	0 a 4.294.967.295
<code>__int64</code>	8	<code>long long</code> , <code>signed long long</code>	- 9.223.372.036.854.775.808 a 9.223.372.036.854.775.807

NOME DO TIPO	BYTES	OUTROS NOMES	INTERVALO DE VALORES
<code>unsigned __int64</code>	8	<code>unsigned long long</code>	0 a 18.446.744.073.709.551.615
<code>bool</code>	1	nenhuma	<code>false</code> or** <code>true</code>
<code>char</code>	1	nenhuma	-128 a 127 por padrão 0 a 255 quando compilado usando /J
<code>signed char</code>	1	nenhuma	-128 a 127
<code>unsigned char</code>	1	nenhuma	0 a 255
<code>short</code>	2	<code>short int</code> , <code>signed short int</code>	-32.768 a 32.767
<code>unsigned short</code>	2	<code>unsigned short int</code>	0 a 65.535
<code>long</code>	4	<code>long int</code> , <code>signed long int</code>	-2.147.483.648 a 2.147.483.647
<code>unsigned long</code>	4	<code>unsigned long int</code>	0 a 4.294.967.295
<code>long long</code>	8	Nenhum (mas equivalente a <code>__int64</code>)	-9.223.372.036.854.775.808 a 9.223.372.036.854.775.807
<code>unsigned long long</code>	8	Nenhum (mas equivalente a <code>unsigned __int64</code>)	0 a 18.446.744.073.709.551.615
<code>enum</code>	varia	nenhuma	
<code>float</code>	4	nenhuma	3.4E +/- 38 (7 dígitos)
<code>double</code>	8	nenhuma	1.7E +/- 308 (15 dígitos)
<code>long double</code>	mesmo que** <code>double</code> **	nenhuma	Mesmo que** <code>double</code> **
<code>wchar_t</code>	2	<code>__wchar_t</code>	0 a 65.535

Dependendo de como é usado, uma variável de `__wchar_t` designa um tipo de caractere largo ou um tipo de caractere multibyte. Use o prefixo `L` antes de uma constante de caractere ou de cadeia de caracteres para designar a constante de tipo de caractere largo.

`signed` e `unsigned` são modificadores que você pode usar com qualquer tipo integral, exceto `bool`. Observe que `char`, `signed char` e `unsigned char` são três tipos distintos para fins de mecanismos, como sobrecarregamentos e modelos.

Os `int` `unsigned int` tipos e têm um tamanho de quatro bytes. No entanto, o código portátil não deve depender do tamanho do `int` porque o padrão de idioma permite que isso seja específico da implementação.

C/C++ no Visual Studio também tem suporte para tipos de inteiros dimensionados. Para obter mais informações, consulte [__int8, __int16, __int32, __int64](#) e [limites de inteiro](#).

Para obter mais informações sobre as restrições dos tamanhos de cada tipo, consulte [tipos internos](#).

O intervalo dos tipos enumerados varia dependendo do contexto de linguagem e dos sinalizadores de compilador especificados. Para obter mais informações, consulte [declarações](#) e [enumerações](#) de [enumeração de C](#).

Confira também

[Palavras-chave](#)

[Tipos internos](#)

nullptr

02/09/2020 • 2 minutes to read • [Edit Online](#)

A `nullptr` palavra-chave especifica uma constante de ponteiro NULL do tipo `std::nullptr_t`, que é conversível para qualquer tipo de ponteiro bruto. Embora você possa usar a palavra-chave `nullptr` sem incluir nenhum cabeçalho, se o seu código usar o tipo `std::nullptr_t`, você deverá defini-lo incluindo o cabeçalho `<cstddef>`.

NOTE

A `nullptr` palavra-chave também é definida em C++/CLI para aplicativos de código gerenciado e não é intercambiável com a palavra-chave ISO Standard C++. Se o código puder ser compilado usando a `/clr` opção do compilador, que tem como alvo o código gerenciado, use `_nullptr` em qualquer linha de código em que você deve garantir que o compilador use a interpretação C++ nativa. Para obter mais informações, consulte [nullptr \(C++/CLI e C++/CX\)](#).

Comentários

Evite usar `NULL` ou zero (`0`) como uma constante de ponteiro NULL; `nullptr` é menos vulnerável ao uso indevido e funciona melhor na maioria das situações. Por exemplo, com `func(std::pair<const char *, double>)`, chamar `func(std::make_pair(NULL, 3.14))` causa um erro do compilador. A macro se `NULL` expande para `0`, de forma que a chamada `std::make_pair(0, 3.14)` retorne `std::pair<int, double>`, que não é conversível para o `std::pair<const char *, double>` tipo de parâmetro no `func`. Chamar `func(std::make_pair(nullptr, 3.14))` resulta em uma compilação bem-sucedida porque `std::make_pair(nullptr, 3.14)` retorna `std::pair<std::nullptr_t, double>`, que é convertido em `std::pair<const char *, double>`.

Confira também

[Palavras-chave](#)

[nullptr \(C++/CLI e C++/CX\)](#)

void (C++)

02/09/2020 • 2 minutes to read • [Edit Online](#)

Quando usado como um tipo de retorno de função, a `void` palavra-chave especifica que a função não retorna um valor. Quando usado para a lista de parâmetros de uma função, `void` especifica que a função não usa nenhum parâmetro. Quando usado na declaração de um ponteiro, `void` especifica que o ponteiro é "Universal".

Se o tipo de um ponteiro `for * void`, o ponteiro poderá apontar para qualquer variável que não esteja declarada com a `const` `volatile` palavra-chave ou. Um **ponteiro * void** não pode ser desreferenciado, a menos que seja convertido em outro tipo. Um **ponteiro * void** pode ser convertido em qualquer outro tipo de ponteiro de dados.

Um `void` ponteiro pode apontar para uma função, mas não para um membro de classe em C++.

Você não pode declarar uma variável do tipo `void`.

Exemplo

```
// void.cpp
void vobject;    // C2182
void *pv;    // okay
int *pint; int i;
int main() {
    pv = &i;
    // Cast optional in C required in C++
    pint = (int *)pv;
}
```

Confira também

[Palavras-chave](#)

[Tipos internos](#)

bool (C++)

02/09/2020 • 2 minutes to read • [Edit Online](#)

Essa palavra-chave é um tipo interno. Uma variável desse tipo pode ter valores `true` e `false`. As expressões condicionais têm o tipo `bool` e, portanto, têm valores do tipo `bool`. Por exemplo, `i != 0` agora tem `true` ou `false` depende do valor de `i`.

Visual Studio 2017 versão 15,3 e posterior (disponível com `/std: c++ 17`): o operando de um sufixo ou incremento de prefixo ou um operador de decréscimo não pode ser do tipo `bool`. Em outras palavras, dada uma variável `b` do tipo `bool`, essas expressões não são mais permitidas:

```
b++;  
++b;  
b--;  
--b;
```

Os valores `true` e `false` têm a seguinte relação:

```
!false == true  
!true == false
```

Na instrução a seguir:

```
if (condexpr1) statement1;
```

Se `condexpr1` for `true`, `statement1` será sempre executado; se `condexpr1` for `false`, `statement1` nunca será executado.

Quando um operador de sufixo ou prefixo `++` é aplicado a uma variável do tipo `bool`, a variável é definida como `true`.

O Visual Studio 2017 versão 15,3 e posteriores: `operator++` para `bool` foi removido do idioma e não tem mais suporte.

O operador sufixo ou prefix `--` não pode ser aplicado a uma variável desse tipo.

O `bool` tipo participa de promoções de integral padrão. Um valor de r-value do tipo `bool` pode ser convertido em um r-value do tipo `int`, com `false` tornar-se zero e `true` se tornar um. Como um tipo distinto, `bool` participa da resolução de sobrecarga.

Confira também

[Palavras-chave](#)
[Tipos internos](#)

false (C++)

02/09/2020 • 2 minutes to read • [Edit Online](#)

A palavra-chave é um dos dois valores de uma variável do tipo `bool` ou uma expressão condicional (uma expressão condicional é agora uma `true` expressão booleana). Por exemplo, se `i` for uma variável do tipo `bool`, a `i = false;` instrução será atribuída a `false` `i`.

Exemplo

```
// bool_false.cpp
#include <stdio.h>

int main()
{
    bool bb = true;
    printf_s("%d\n", bb);
    bb = false;
    printf_s("%d\n", bb);
}
```

```
1
0
```

Confira também

[Palavras-chave](#)

true (C++)

02/09/2020 • 2 minutes to read • [Edit Online](#)

Sintaxe

```
bool-identifier = true ;
bool-expression logical-operator true ;
```

Comentários

Essa palavra-chave é um dos dois valores de uma variável do tipo `bool` ou uma expressão condicional (uma expressão condicional é agora uma expressão booleana verdadeira). Se `i` for do tipo `bool`, a instrução será `i = true;` atribuída a `true` `i`.

Exemplo

```
// bool_true.cpp
#include <stdio.h>
int main()
{
    bool bb = true;
    printf_s("%d\n", bb);
    bb = false;
    printf_s("%d\n", bb);
}
```

```
1
0
```

Confira também

[Palavras-chave](#)

char, wchar_t, char16_t, char32_t

02/09/2020 • 3 minutes to read • [Edit Online](#)

Os tipos `char`, `wchar_t`, `char16_t` e `char32_t` são tipos internos que representam caracteres alfanuméricos, bem como glifos não alfanuméricos e caracteres não imprimíveis.

Sintaxe

```
char      ch1{ 'a' }; // or { u8'a' }
wchar_t   ch2{ L'a' };
char16_t  ch3{ u'a' };
char32_t  ch4{ U'a' };
```

Comentários

O `char` tipo era o tipo de caractere original em C e C++. O tipo `unsigned char` é geralmente usado para representar um *byte*, que não é um tipo interno em C++. O `char` tipo pode ser usado para armazenar caracteres do conjunto de caracteres ASCII ou de qualquer um dos conjuntos de caracteres ISO-8859 e bytes individuais de caracteres de byte múltiplo, como Shift-JIS ou a codificação UTF-8 do conjunto de caracteres Unicode. As cadeias de caracteres do `char` tipo são chamadas de cadeias *restritas*, mesmo quando usadas para codificar os personagens com vários bytes. No compilador Microsoft, `char` é um tipo de 8 bits.

O `wchar_t` tipo é um tipo de caractere largo definido pela implementação. No compilador da Microsoft, ele representa um caractere largo de 16 bits usado para armazenar o Unicode codificado como UTF-16LE, o tipo de caractere nativo em sistemas operacionais Windows. As versões de caracteres largos das funções de biblioteca de tempo de execução universal C (UCRT) usam `wchar_t` e seus tipos de ponteiro e de matriz como parâmetros e valores de retorno, assim como as versões de caracteres largos da API nativa do Windows.

Os `char16_t` `char32_t` tipos e representam caracteres largos de 16 e 32 bits, respectivamente. O Unicode codificado como UTF-16 pode ser armazenado no `char16_t` tipo, e o Unicode codificado como UTF-32 pode ser armazenado no `char32_t` tipo. Cadeias desses tipos e `wchar_t` são todas conhecidas como cadeias de caracteres *largos*, embora o termo geralmente se refira especificamente a cadeias de caracteres do `wchar_t` tipo.

Na biblioteca C++ Standard, o `basic_string` tipo é especializado para cadeias de caracteres estreitas e largas. Use `std::string` quando os caracteres forem do tipo `char`, `std::u16string` quando os caracteres forem do tipo `char16_t`, `std::u32string` quando os caracteres forem do tipo `char32_t` e `std::wstring` quando os caracteres forem do tipo `wchar_t`. Outros tipos que representam texto, incluindo `std::stringstream` e `std::cout` têm especializações para cadeias de caracteres estreitas e largos.

__int8, __int16, __int32, __int64

02/09/2020 • 2 minutes to read • [Edit Online](#)

Específico da Microsoft

O Microsoft C/C++ tem suporte para tipos de inteiros dimensionados. Você pode declarar variáveis de inteiro de 8, 16, 32-ou 64 bits usando o `__intN` especificador de tipo, em que `N` é 8, 16, 32 ou 64.

O exemplo a seguir declara uma variável para cada um desses tipos de inteiros dimensionados:

```
__int8 nSmall;      // Declares 8-bit integer
__int16 nMedium;    // Declares 16-bit integer
__int32 nLarge;     // Declares 32-bit integer
__int64 nHuge;      // Declares 64-bit integer
```

Os tipos `__int8`, `__int16`, e `__int32` são sinônimos para os tipos ANSI que têm o mesmo tamanho e são úteis para escrever código portável que se comporta de forma idêntica em várias plataformas. O `__int8` tipo de dados é sinônimo de tipo `char`, `__int16` é sinônimo de tipo `short` e `__int32` é sinônimo de tipo `int`. O `__int64` tipo é sinônimo de tipo `long long`.

Para compatibilidade com versões anteriores, `__int8`, `__int16`, `__int32` e `__int64` são sinônimos para `__int8`, `__int16`, `__int32`, e `__int64` a menos que a opção do compilador [/za](#) (desabilite extensões de linguagem) seja especificada.

Exemplo

O exemplo a seguir mostra que um `__intN` parâmetro será promovido para `int`:

```
// sized_int_types.cpp

#include <stdio.h>

void func(int i) {
    printf_s("%s\n", __FUNCTION__);
}

int main()
{
    __int8 i8 = 100;
    func(i8);    // no void func(__int8 i8) function
                 // __int8 will be promoted to int
}
```

```
func
```

Confira também

[Palavras-chave](#)

[Tipos internos](#)

[Intervalos de tipos de dados](#)

__m64

02/09/2020 • 2 minutes to read • [Edit Online](#)

Específico da Microsoft

O __m64 tipo de dados é para uso com a tecnologia MMX e 3DNow! intrínsecos e é definido em <xmmmintrin.h> .

```
// data_types__m64.cpp
#include <xmmmintrin.h>
int main()
{
    __m64 x;
}
```

Comentários

Você não deve acessar os __m64 campos diretamente. No entanto, você pode consultar esses tipos no depurador.

Uma variável do tipo é __m64 mapeada para os registros mm [0-7].

Variáveis do tipo __m64 são alinhadas automaticamente em limites de 8 bytes.

__m64 Não há suporte para o tipo de dados em processadores x64. Os aplicativos que usam __m64 como parte de intrínsecos de MMX devem ser reescritos para usar intrínsecos de SSE e SSE2.

FINAL específico da Microsoft

Confira também

[Palavras-chave](#)

[Tipos internos](#)

[Intervalos de tipos de dados](#)

__m128

02/09/2020 • 2 minutes to read • [Edit Online](#)

Específico da Microsoft

O tipo de **dados** `__m128` , para uso com as instruções Streaming SIMD Extensions e Streaming SIMD Extensions 2, são intrínsecas, é definido em `<xmmmintrin.h>` .

```
// data_types__m128.cpp
#include <xmmmintrin.h>
int main() {
    __m128 x;
}
```

Comentários

Você não deve acessar os `__m128` campos diretamente. No entanto, você pode consultar esses tipos no depurador. Uma variável do tipo é `__m128` mapeada para os registros de XMM [0-7].

As variáveis do tipo `__m128` são alinhadas automaticamente em limites de 16 bytes.

`__m128` Não há suporte para o tipo de dados em processadores ARM.

FINAL específico da Microsoft

Confira também

[Palavras-chave](#)

[Tipos internos](#)

[Intervalos de tipos de dados](#)

__m128d

02/09/2020 • 2 minutes to read • [Edit Online](#)

Específico da Microsoft

O `__m128d` tipo de dados, para uso com as instruções de Streaming SIMD Extensions 2, é intrínseco, é definido em `<emmintrin.h>`.

```
// data_types__m128d.cpp
#include <emmintrin.h>
int main() {
    __m128d x;
}
```

Comentários

Você não deve acessar os `__m128d` campos diretamente. No entanto, você pode consultar esses tipos no depurador. Uma variável do tipo é `__m128` mapeada para os registros de XMM [0-7].

Variáveis do tipo `__m128d` são alinhadas automaticamente em limites de 16 bytes.

`__m128d` Não há suporte para o tipo de dados em processadores ARM.

FINAL específico da Microsoft

Confira também

[Palavras-chave](#)

[Tipos internos](#)

[Intervalos de tipos de dados](#)

__m128i

02/09/2020 • 2 minutes to read • [Edit Online](#)

Específico da Microsoft

O `__m128i` tipo de dados, para uso com as instruções SSE2 (Streaming SIMD Extensions 2) é intrínseco, é definido em `<emmintrin.h>`.

```
// data_types__m128i.cpp
#include <emmintrin.h>
int main() {
    __m128i x;
}
```

Comentários

Você não deve acessar os `__m128i` campos diretamente. No entanto, você pode consultar esses tipos no depurador. Uma variável do tipo é `__m128i` mapeada para os registros de XMM [0-7].

As variáveis do tipo `__m128i` são alinhadas automaticamente em limites de 16 bytes.

NOTE

O uso de variáveis do tipo `__m128i` fará com que o compilador gere a `movdqa` instrução SSE2. Essa instrução não causa uma falha em processadores Pentium III, mas resultará em falha silenciosa, com possíveis efeitos colaterais causados por quaisquer instruções que se `movdqa` traduzem em processadores Pentium III.

`__m128i` Não há suporte para o tipo de dados em processadores ARM.

FINAL específico da Microsoft

Confira também

[Palavras-chave](#)

[Tipos internos](#)

[Intervalos de tipos de dados](#)

__ptr32, __ptr64

02/09/2020 • 2 minutes to read • [Edit Online](#)

Específico da Microsoft

`__ptr32` representa um ponteiro nativo em um sistema de 32 bits, enquanto `__ptr64` representa um ponteiro nativo em um sistema de 64 bits.

O exemplo a seguir mostra como declarar cada um desses tipos de ponteiro:

```
int * __ptr32 p32;
int * __ptr64 p64;
```

Em um sistema de 32 bits, um ponteiro declarado com `__ptr64` é truncado para um ponteiro de 32 bits. Em um sistema de 64 bits, um ponteiro declarado com `__ptr32` é forçado a um ponteiro de 64 bits.

NOTE

Você não pode usar `__ptr32` ou `__ptr64` ao compilar com `/CLR: Pure`. Caso contrário, o erro do compilador C2472 será gerado. As opções de compilador `/CLR: Pure` e `/CLR: safe` são preteridas no Visual Studio 2015 e sem suporte no Visual Studio 2017.

Para compatibilidade com versões anteriores, `__ptr32` e `__ptr64` são sinônimos de `__ptr32` e `__ptr64`, a menos que a opção do compilador `/za (desabilitar extensões de idioma)` seja especificada.

Exemplo

O exemplo a seguir mostra como declarar e alocar ponteiros com as `__ptr32` `__ptr64` palavras-chave e.

```
#include <cstdlib>
#include <iostream>

int main()
{
    using namespace std;

    int * __ptr32 p32;
    int * __ptr64 p64;

    p32 = (int * __ptr32)malloc(4);
    *p32 = 32;
    cout << *p32 << endl;

    p64 = (int * __ptr64)malloc(4);
    *p64 = 64;
    cout << *p64 << endl;
}
```

32
64

Confira também

[Tipos internos](#)

Limites numéricos (C++)

25/03/2020 • 2 minutes to read • [Edit Online](#)

Os dois arquivos de inclusão padrão, `<Limits.h>` e `<float.h>`, definem os limites numéricos ou os valores mínimo e máximo que uma variável de um determinado tipo pode conter. Esses mínimos e máximos são garantidos como portáteis para qualquer C++ compilador que usa a mesma representação de dados que o ANSI C. O `<Limits.h>` arquivo de inclusão define os [limites numéricos para tipos inteiros](#) e `<float.h>` define os [limites numéricos para tipos flutuantes](#).

Confira também

[Conceitos básicos](#)

Limites de inteiro

02/09/2020 • 2 minutes to read • [Edit Online](#)

Específico da Microsoft

Os limites para tipos de inteiros estão listadas na tabela a seguir. As macros de pré-processador para esses limites também são definidas quando você inclui o arquivo de cabeçalho padrão `<climits>`.

Limites em constantes de inteiro

CONSTANTE	SIGNIFICADO	VALOR
<code>CHAR_BIT</code>	Número de bits na menor variável que não é um campo de bit.	8
<code>SCHAR_MIN</code>	Valor mínimo para uma variável do tipo <code>signed char</code> .	-128
<code>SCHAR_MAX</code>	Valor máximo para uma variável do tipo <code>signed char</code> .	127
<code>UCHAR_MAX</code>	Valor máximo para uma variável do tipo <code>unsigned char</code> .	255 (0xff)
<code>CHAR_MIN</code>	Valor mínimo para uma variável do tipo <code>char</code> .	-128; 0 se a <code>/J</code> opção for usada
<code>CHAR_MAX</code>	Valor máximo para uma variável do tipo <code>char</code> .	127; 255 se a <code>/J</code> opção for usada
<code>MB_LEN_MAX</code>	Número máximo de bytes em uma constante de vários caracteres.	5
<code>SHRT_MIN</code>	Valor mínimo para uma variável do tipo <code>short</code> .	-32768
<code>SHRT_MAX</code>	Valor máximo para uma variável do tipo <code>short</code> .	32767
<code>USHRT_MAX</code>	Valor máximo para uma variável do tipo <code>unsigned short</code> .	65535 (0xffff)
<code>INT_MIN</code>	Valor mínimo para uma variável do tipo <code>int</code> .	-2147483648
<code>INT_MAX</code>	Valor máximo para uma variável do tipo <code>int</code> .	2147483647
<code>UINT_MAX</code>	Valor máximo para uma variável do tipo <code>unsigned int</code> .	4294967295 (0xffffffff)

CONSTANTE	SIGNIFICADO	VALOR
<code>LONG_MIN</code>	Valor mínimo para uma variável do tipo <code>long</code> .	-2147483648
<code>LONG_MAX</code>	Valor máximo para uma variável do tipo <code>long</code> .	2147483647
<code>ULONG_MAX</code>	Valor máximo para uma variável do tipo <code>unsigned long</code> .	4294967295 (0xffffffff)
<code>LLONG_MIN</code>	Valor mínimo para uma variável do tipo** <code>long long</code> **	-9223372036854775808
<code>LLONG_MAX</code>	Valor máximo para uma variável do tipo** <code>long long</code> **	9223372036854775807
<code>ULLONG_MAX</code>	Valor máximo para uma variável do tipo** <code>unsigned long long</code> **	18446744073709551615 (0xffffffffffffffffffff)

Se um valor exceder a representação do maior inteiro, o compilador da Microsoft gera um erro.

Confira também

[Limites flutuantes](#)

Limites flutuantes

15/04/2020 • 2 minutes to read • [Edit Online](#)

Específico da Microsoft

A tabela a seguir lista os limites nos valores de constantes de ponto flutuante. Esses limites também são definidos <no arquivo de cabeçalho padrão float.h>.

Limites em constantes de ponto flutuante

CONSTANTE	SIGNIFICADO	VALOR
<code>FLT_DIG</code> <code>DBL_DIG</code> <code>LDBL_DIG</code>	Número de dígitos, q , de modo que um número de ponto flutuante com dígitos decimais de q possam ser arredondados em uma representação de ponto flutuante e de volta sem perda de precisão.	6 15 15
<code>FLT_EPSILON</code> <code>DBL_EPSILON</code> <code>LDBL_EPSILON</code>	O menor número positivo x , de modo que $x + 1,0$ não é igual a $1,0$.	1.192092896e-07F 2.2204460492503131e-016 2.2204460492503131e-016
<code>FLT_GUARD</code>		0
<code>FLT_MANT_DIG</code> <code>DBL_MANT_DIG</code> <code>LDBL_MANT_DIG</code>	Número de dígitos no radix <code>FLT_RADIX</code> especificado pelo ponto flutuante. O radix é 2; portanto, esses valores especificam bits.	24 53 53
<code>FLT_MAX</code> <code>DBL_MAX</code> <code>LDBL_MAX</code>	Número máximo de ponto flutuante representando.	3.402823466e+38F 1,7976931348623158e+308 1,7976931348623158e+308
<code>FLT_MAX_10_EXP</code> <code>DBL_MAX_10_EXP</code> <code>LDBL_MAX_10_EXP</code>	O inteiro máximo de tal forma que 10 elevados a esse número é um número de ponto flutuante representando.	38 308 308
<code>FLT_MAX_EXP</code> <code>DBL_MAX_EXP</code> <code>LDBL_MAX_EXP</code>	O inteiro máximo <code>FLT_RADIX</code> que elevou para esse número é um número de ponto flutuante representando.	128 1024 1024
<code>FLT_MIN</code> <code>DBL_MIN</code> <code>LDBL_MIN</code>	Valor positivo mínimo.	1.175494351e-38F 2,2250738585072014e-308 2,2250738585072014e-308
<code>FLT_MIN_10_EXP</code> <code>DBL_MIN_10_EXP</code> <code>LDBL_MIN_10_EXP</code>	O inteiro negativo mínimo de tal forma que 10 elevados a esse número é um número de ponto flutuante representando.	-37 -307 -307

CONSTANTE	SIGNIFICADO	VALOR
<code>FLT_MIN_EXP</code> <code>DBL_MIN_EXP</code> <code>LDBL_MIN_EXP</code>	O inteiro negativo mínimo <code>FLT_RADIX</code> que elevou para esse número é um número de ponto flutuante representando.	-125 -1021 -1021
<code>FLT_NORMALIZE</code>		0
<code>FLT_RADIX</code> <code>_DBL_RADIX</code> <code>_LDBL_RADIX</code>	Raiz de representação do expoente.	2 2 2
<code>FLT_ROUNDS</code> <code>_DBL_ROUNDS</code> <code>_LDBL_ROUNDS</code>	Modo de arredondamento para adição de ponto flutuante.	1 (próximo) 1 (próximo) 1 (próximo)

NOTE

As informações da tabela podem ser diferente em versões futuras do produto.

Fim específico da Microsoft

Confira também

[Limites inteiros](#)

Declarações e definições (C++)

02/09/2020 • 8 minutes to read • [Edit Online](#)

Um programa C++ consiste em várias entidades, como variáveis, funções, tipos e namespaces. Cada uma dessas entidades deve ser *declarada* antes que possa ser usada. Uma declaração especifica um nome exclusivo para a entidade, juntamente com informações sobre seu tipo e outras características. Em C++, o ponto no qual um nome é declarado é o ponto em que ele se torna visível para o compilador. Não é possível fazer referência a uma função ou classe declarada em algum momento posterior na unidade de compilação. As variáveis devem ser declaradas o mais próximo possível antes do ponto em que são usadas.

O exemplo a seguir mostra algumas declarações:

```
#include <string>

void f(); // forward declaration

int main()
{
    const double pi = 3.14; //OK
    int i = f(2); //OK. f is forward-declared
    std::string str; // OK std::string is declared in <string> header
    C obj; // error! C not yet declared.
    j = 0; // error! No type specified.
    auto k = 0; // OK. type inferred as int by compiler.
}

int f(int i)
{
    return i + 42;
}

namespace N {
    class C{/*...*/};
}
```

Na linha 5, a `main` função é declarada. Na linha 7, uma `const` variável chamada `pi` é declarada e *inicializada*. Na linha 8, um inteiro `i` é declarado e inicializado com o valor produzido pela função `f`. O nome `f` é visível para o compilador devido à *declaração de encaminhamento* na linha 3.

Na linha 9, uma variável chamada `obj` do tipo `C` é declarada. No entanto, essa declaração gera um erro porque `C` não é declarada até mais tarde no programa e não é declarada de encaminhamento. Para corrigir o erro, você pode mover toda a *definição* de `C` antes `main` ou adicionar uma declaração de encaminhamento para ele. Esse comportamento é diferente de outras linguagens, como C#, em que funções e classes podem ser usadas antes de seu ponto de declaração em um arquivo de origem.

Na linha 10, uma variável chamada `str` do tipo `std::string` é declarada. O nome `std::string` é visível porque é introduzido no arquivo de `string` *cabeçalho* que é mesclado no arquivo de origem na linha 1. `std` é o namespace no qual a `string` classe é declarada.

Na linha 11, um erro é gerado porque o nome `j` não foi declarado. Uma declaração deve fornecer um tipo, diferente de outras linguagens, como JavaScript. Na linha 12, a `auto` palavra-chave é usada, o que informa ao compilador para inferir o tipo de `k` com base no valor com o qual ele foi inicializado. O compilador, nesse caso, escolhe `int` para o tipo.

Escopo de declaração

O nome que é introduzido por uma declaração é válido dentro do *escopo* onde a declaração ocorre. No exemplo anterior, as variáveis que são declaradas dentro da `main` função são *variáveis locais*. Você pode declarar outra variável chamada `i` fora do principal, no *escopo global*, e seria uma entidade completamente separada. No entanto, essa duplicação de nomes pode levar à confusão e aos erros do programador e deve ser evitada. Na linha 21, a classe `C` é declarada no escopo do namespace `N`. O uso de namespaces ajuda a evitar *colisões de nomes*. A maioria dos nomes de biblioteca padrão do C++ são declarados dentro do `std` namespace. Para obter mais informações sobre como as regras de escopo interagem com declarações, consulte [escopo](#).

Definições

Algumas entidades, incluindo funções, classes, enums e variáveis constantes, devem ser definidas além de serem declaradas. Uma *definição* fornece ao compilador todas as informações necessárias para gerar o código do computador quando a entidade é usada posteriormente no programa. No exemplo anterior, a linha 3 contém uma declaração para a função `f`, mas a *definição* da função é fornecida nas linhas de 15 a 18. Na linha 21, a classe `C` é declarada e definida (embora conforme definido, a classe não faz nada). Uma variável Constant deve ser definida, em outras palavras, um valor atribuído, na mesma instrução em que é declarada. Uma declaração de um tipo interno, como `int` é automaticamente uma definição, porque o compilador sabe quanto espaço deve ser alocado para ele.

O exemplo a seguir mostra declarações que também são definições:

```
// Declare and define int variables i and j.
int i;
int j = 10;

// Declare enumeration suits.
enum suits { Spades = 1, Clubs, Hearts, Diamonds };

// Declare class CheckBox.
class CheckBox : public Control
{
public:
    Boolean IsChecked();
    virtual int ChangeState() = 0;
};
```

Aqui estão algumas declarações que não são definições:

```
extern int i;
char *strchr( const char *Str, const char Target );
```

TYPEDEFs e instruções using

Em versões mais antigas do C++, a `typedef` palavra-chave é usada para declarar um novo nome que é um *alias* para outro nome. Por exemplo, o tipo `std::string` é outro nome para `std::basic_string<char>`. Deve ser óbvio por que os programadores usam o nome do `typedef` e não o nome real. No C++ moderno, a `using` palavra-chave é preferida `typedef`, mas a ideia é a mesma: um novo nome é declarado para uma entidade que já está declarada e definida.

Membros de classe estática

Como os membros de dados de classe estática são variáveis discretas compartilhadas por todos os objetos da classe, eles devem ser definidos e inicializados fora da definição de classe. (Para obter mais informações, consulte

classes.)

declarações extern

Um programa C++ pode conter mais de uma [unidade de compilação](#). Para declarar uma entidade que é definida em uma unidade de compilação separada, use a `extern` palavra-chave. As informações na declaração são suficientes para o compilador, mas se a definição da entidade não puder ser encontrada na etapa de vinculação, o vinculador gerará um erro.

Nesta seção

[Classes de armazenamento](#)

`const`

`constexpr`

`extern`

[Inicializadores](#)

[Aliases e typedefs](#)

`using` `mesma`

`volatile`

`decltype`

[Atributos em C++](#)

Confira também

[Conceitos básicos](#)

Classes de armazenamento

02/09/2020 • 13 minutes to read • [Edit Online](#)

Uma *classe de armazenamento* no contexto de declarações de variáveis de C++ é um especificador de tipo que governa o tempo de vida, a vinculação e o local da memória dos objetos. Um determinado objeto pode ter apenas uma classe de armazenamento. As variáveis definidas em um bloco têm armazenamento automático, a menos que especificado de outra forma usando os `extern` `static` `thread_local` especificadores, ou. Objetos e variáveis automáticas não têm vínculos; Eles não são visíveis para o código fora do bloco. A memória é alocada para elas automaticamente quando a execução entra no bloco e desalocada quando o bloco é encerrado.

Observações

1. A palavra-chave `mutável` pode ser considerada um especificador de classe de armazenamento. No entanto, ela só está disponível na lista de membros de uma definição de classe.
2. **Visual Studio 2010 e posterior:** A `auto` palavra-chave não é mais um especificador de classe de armazenamento C++ e a `register` palavra-chave é preterida. **Visual Studio 2017 versão 15,7 e posterior:** (disponível com `/std:c++17`): a `register` palavra-chave é removida da linguagem C++.

```
register int val; // warning C5033: 'register' is no longer a supported storage class
```

static

A `static` palavra-chave pode ser usada para declarar variáveis e funções em escopo global, escopo de namespace e escopo de classe. Variáveis estáticas também podem ser declaradas no escopo local.

Duração estática significa que o objeto ou a variável são alocados quando o programa inicia e desalocados quando o programa termina. Vinculação externa significa que o nome da variável é visível fora do arquivo em que a variável é declarada. Por outro lado, a vinculação externa significa que o nome não é visível fora do arquivo em que a variável é declarada. Por padrão, um objeto ou uma variável que é definida no namespace global tem duração estática e ligação externa. A `static` palavra-chave pode ser usada nas situações a seguir.

1. Quando você declara uma variável ou função no escopo do arquivo (escopo global e/ou namespace), a `static` palavra-chave especifica que a variável ou função tem vínculo interno. Ao declarar uma variável, a variável tem duração estática, e o compilador a inicializa com o valor 0 a menos que você especifique outro valor.
2. Quando você declara uma variável em uma função, a `static` palavra-chave especifica que a variável retém seu estado entre chamadas para essa função.
3. Quando você declara um membro de dados em uma declaração de classe, a `static` palavra-chave especifica que uma cópia do membro é compartilhada por todas as instâncias da classe. Um membro de dados `static` deve ser definido no escopo do arquivo. Um membro de dados integral que você declara como `const static` pode ter um inicializador.
4. Quando você declara uma função de membro em uma declaração de classe, a `static` palavra-chave especifica que a função é compartilhada por todas as instâncias da classe. Uma função de membro `static` não pode acessar um membro de instância porque a função não tem um `this` ponteiro implícito. Para acessar um membro de instância, declare a função com um parâmetro que seja um ponteiro ou uma referência de instância.

5. Não é possível declarar os membros de uma união como estáticos. No entanto, uma União anônima declarada globalmente deve ser declarada explicitamente `static`.

Este exemplo mostra como uma variável declarada `static` em uma função retém seu estado entre chamadas para essa função.

```
// static1.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
void showstat( int curr ) {
    static int nStatic;    // Value of nStatic is retained
                           // between each function call
    nStatic += curr;
    cout << "nStatic is " << nStatic << endl;
}

int main() {
    for ( int i = 0; i < 5; i++ )
        showstat( i );
}
```

```
nStatic is 0
nStatic is 1
nStatic is 3
nStatic is 6
nStatic is 10
```

Este exemplo mostra o uso de `static` em uma classe.

```
// static2.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
class CMyClass {
public:
    static int m_i;
};

int CMyClass::m_i = 0;
CMyClass myObject1;
CMyClass myObject2;

int main() {
    cout << myObject1.m_i << endl;
    cout << myObject2.m_i << endl;

    myObject1.m_i = 1;
    cout << myObject1.m_i << endl;
    cout << myObject2.m_i << endl;

    myObject2.m_i = 2;
    cout << myObject1.m_i << endl;
    cout << myObject2.m_i << endl;

    CMyClass::m_i = 3;
    cout << myObject1.m_i << endl;
    cout << myObject2.m_i << endl;
}
```

```
0
0
1
1
2
2
3
3
```

Este exemplo mostra uma variável local declarada `static` em uma função de membro. A `static` variável está disponível para todo o programa; todas as instâncias do tipo compartilham a mesma cópia da `static` variável.

```
// static3.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;
struct C {
    void Test(int value) {
        static int var = 0;
        if (var == value)
            cout << "var == value" << endl;
        else
            cout << "var != value" << endl;

        var = value;
    }
};

int main() {
    C c1;
    C c2;
    c1.Test(100);
    c2.Test(100);
}
```

```
var != value
var == value
```

A partir do C++ 11, `static` é garantido que a inicialização de uma variável local seja thread-safe. Esse recurso é, às vezes, chamado de *estáticos mágicos*. No entanto, em um aplicativo multithread, todas as atribuições subsequentes devem ser sincronizadas. O recurso de inicialização estática thread-safe pode ser desabilitado usando o `/Zc:threadSafeInit-` sinalizador para evitar a obtenção de uma dependência no CRT.

extern

Objetos e variáveis declarados como `extern` declara um objeto que é definido em outra unidade de tradução ou em um escopo delimitador como tendo vínculo externo. Para obter mais informações, consulte [extern](#) e [unidades de tradução e ligação](#).

thread_local (C++ 11)

Uma variável declarada com o `thread_local` especificador é acessível somente no thread no qual ele é criado. A variável é criada quando o thread é criado e destruída quando o thread é destruído. Cada thread tem sua própria cópia da variável. No Windows, o `thread_local` é funcionalmente equivalente ao atributo específico da Microsoft `__declspec(thread)`.

```

thread_local float f = 42.0; // Global namespace. Not implicitly static.

struct S // cannot be applied to type definition
{
    thread_local int i; // Illegal. The member must be static.
    thread_local static char buf[10]; // OK
};

void DoSomething()
{
    // Apply thread_local to a local variable.
    // Implicitly "thread_local static S my_struct".
    thread_local S my_struct;
}

```

Itens a serem observados sobre o `thread_local` especificador:

- Variáveis locais de thread inicializadas dinamicamente em DLLs podem não ser inicializadas corretamente em todos os threads de chamada. Para obter mais informações, consulte [thread](#).
- O `thread_local` especificador pode ser combinado com `static` ou `extern`.
- Você pode aplicar `thread_local` somente a definições e declarações de dados; `thread_local` não pode ser usado em declarações ou definições de função.
- Você pode especificar `thread_local` somente em itens de dados com duração de armazenamento estático. Isso inclui objetos de dados globais (`static` e `extern`), objetos estáticos locais e membros de dados estáticos de classes. Qualquer variável local declarada `thread_local` será implicitamente estática se nenhuma outra classe de armazenamento for fornecida; em outras palavras, no escopo do bloco `thread_local` será equivalente a `thread_local static`.
- Você deve especificar `thread_local` para a declaração e a definição de um objeto local de thread, se a declaração e a definição ocorrem no mesmo arquivo ou em arquivos separados.

No Windows, `thread_local` é funcionalmente equivalente a `__declspec(thread)`, exceto que `*__declspec(thread)` * pode ser aplicado a uma definição de tipo e é válido no código C. Sempre que possível, use `thread_local` porque ele faz parte do C++ padrão e, portanto, é mais portável.

Registr

Visual Studio 2017 versão 15,3 e posterior (disponível com `/std:c++17`): a `register` palavra-chave não é mais uma classe de armazenamento com suporte. A palavra-chave ainda está reservada no padrão para uso futuro.

```
register int val; // warning C5033: 'register' is no longer a supported storage class
```

Exemplo: inicialização automática vs. estática

Um objeto ou uma variável local automática são inicializados cada vez que o fluxo de controle alcança sua definição. Um objeto ou uma variável local estática são inicializados na primeira vez que o fluxo de controle alcança sua definição.

Considere o exemplo a seguir, que define uma classe que registra a inicialização e a destruição de objetos e depois define três objetos, `I1`, `I2` e `I3`:

```

// initialization_of_objects.cpp
// compile with: /EHsc
#include <iostream>
#include <string.h>
using namespace std;

// Define a class that logs initializations and destructions.
class InitDemo {
public:
    InitDemo( const char *szWhat );
    ~InitDemo();

private:
    char *szObjName;
    size_t sizeofObjName;
};

// Constructor for class InitDemo
InitDemo::InitDemo( const char *szWhat ) :
szObjName(NULL), sizeofObjName(0) {
if ( szWhat != 0 && strlen( szWhat ) > 0 ) {
    // Allocate storage for szObjName, then copy
    // initializer szWhat into szObjName, using
    // secured CRT functions.
    sizeofObjName = strlen( szWhat ) + 1;

    szObjName = new char[ sizeofObjName ];
    strcpy_s( szObjName, sizeofObjName, szWhat );

    cout << "Initializing: " << szObjName << "\n";
}
else {
    szObjName = 0;
}
}

// Destructor for InitDemo
InitDemo::~InitDemo() {
if( szObjName != 0 ) {
    cout << "Destroying: " << szObjName << "\n";
    delete szObjName;
}
}

// Enter main function
int main() {
    InitDemo I1( "Auto I1" );
    cout << "In block.\n";
    InitDemo I2( "Auto I2" );
    static InitDemo I3( "Static I3" );
}
cout << "Exited block.\n";
}

```

```

Initializing: Auto I1
In block.
Initializing: Auto I2
Initializing: Static I3
Destroying: Auto I2
Exited block.
Destroying: Auto I1
Destroying: Static I3

```

Este exemplo demonstra como e quando os objetos `I1`, `I2` e `I3` são inicializados e quando são destruídos.

Há vários pontos a serem observados sobre o programa:

- Primeiro, `I1` e `I2` são destruídos automaticamente quando o fluxo de controle sai do bloco em que estão definidos.
- Depois, em C++, não é necessário declarar objetos ou variáveis no início de um bloco. Além disso, esses objetos são inicializados somente quando o fluxo de controle atinge suas definições. (`I2` e `I3` são exemplos dessas definições.) A saída mostra exatamente quando elas são inicializadas.
- Por fim, as variáveis locais estáticas, como `I3`, retêm seus valores enquanto o programa dura, mas são destruídas quando o programa é encerrado.

Confira também

[Declarações e definições](#)

Deduz o tipo de uma variável declarada da expressão de inicialização.

NOTE

O padrão C++ define um significado original e revisado para essa palavra-chave. Antes do Visual Studio 2010, a `auto` palavra-chave declara uma variável na classe de armazenamento *automática*; ou seja, uma variável que tem um tempo de vida local. A partir do Visual Studio 2010, a `auto` palavra-chave declara uma variável cujo tipo é deduzido da expressão de inicialização em sua declaração. A opção de compilador `/Zc:auto` controla o significado da `auto` palavra-chave.

Sintaxe

```
auto ***inicializador* de *Declarador*** ;
```

```
[](auto ***param1* , auto *param2*** ) {};
```

Comentários

A `auto` palavra-chave direciona o compilador para usar a expressão de inicialização de uma variável declarada, ou parâmetro de expressão lambda, para deduzir seu tipo.

Recomendamos que você use a `auto` palavra-chave para a maioria das situações — a menos que você realmente queira uma conversão — porque ela fornece estes benefícios:

- **Robustez:** Se o tipo da expressão for alterado — isso incluirá quando um tipo de retorno de função for alterado — ele simplesmente funcionará.
- **Desempenho:** Você tem a garantia de que não haverá nenhuma conversão.
- **Usabilidade:** Você não precisa se preocupar com as dificuldades e erros de ortografia do nome do tipo.
- **Eficiência:** Sua codificação pode ser mais eficiente.

Casos de conversão nos quais você talvez não queira usar `auto`:

- Quando você desejar um tipo específico e nada mais serve.
- Tipos de auxiliares de modelo de expressão — por exemplo, `(valarray+valarray)`.

Para usar a `auto` palavra-chave, use-a em vez de um tipo para declarar uma variável e especifique uma expressão de inicialização. Além disso, você pode modificar a `auto` palavra-chave usando especificadores e declaradores como `const`, `volatile`, ponteiro (`*`), referência (`()`) e `&` referência de rvalue (`&&`). O compilador avalia a expressão e de inicialização e usa essas informações para deduzir o tipo da variável.

A expressão de inicialização pode ser uma atribuição (sintaxe de sinal de igual), uma inicialização direta (sintaxe de estilo de função), uma `operator new` expressão ou a expressão de inicialização pode ser o parâmetro *para declaração de intervalo* em uma instrução de `instrução baseada em intervalo for` (C++). Para obter mais informações, consulte [inicializadores](#) e os exemplos de código mais adiante neste documento.

A `auto` palavra-chave é um espaço reservado para um tipo, mas não é um tipo. Portanto, a `auto` palavra-chave

não pode ser usada em conversões ou operadores como `sizeof` e (para C++/CLI) `typeid` .

Utilidade

A `auto` palavra-chave é uma maneira simples de declarar uma variável que tem um tipo complicado. Por exemplo, você pode usar `auto` para declarar uma variável em que a expressão de inicialização envolve modelos, ponteiros para funções ou ponteiros para membros.

Você também pode usar `auto` para declarar e inicializar uma variável para uma expressão lambda. Você não pode declarar o tipo da variável você mesmo pois o tipo de uma expressão lambda é conhecido apenas pelo compilador. Para obter mais informações, consulte [exemplos de expressões lambda](#).

Rastreamento de tipos de retorno

Você pode usar `auto`, junto com o `decltype` especificador de tipo, para ajudar a gravar bibliotecas de modelos. Use `auto` e `decltype` para declarar uma função de modelo cujo tipo de retorno depende dos tipos de seus argumentos de modelo. Ou use `auto` e `decltype` para declarar uma função de modelo que encapsula uma chamada para outra função e, em seguida, retorna qualquer que seja o tipo de retorno dessa outra função. Para obter mais informações, consulte [decltype](#).

Referências e qualificadores CV

Observe que o uso `auto` de referências de Descartes, `const` qualificadores e `volatile` qualificadores.

Considere o exemplo a seguir:

```
// cl.exe /analyze /EHsc /W4
#include <iostream>

using namespace std;

int main( )
{
    int count = 10;
    int& countRef = count;
    auto myAuto = countRef;

    countRef = 11;
    cout << count << " ";

    myAuto = 12;
    cout << count << endl;
}
```

No exemplo anterior, `myAuto` é um `int`, não uma `int` referência, portanto, a saída é `11 11`, não `11 12` como seria o caso se o qualificador de referência não tivesse sido descartado pelo `auto` .

Dedução de tipo com inicializadores de chaves (C++ 14)

O exemplo de código a seguir mostra como inicializar uma `auto` variável usando chaves. Observe a diferença entre B e C e entre A e e.

```

#include <initializer_list>

int main()
{
    // std::initializer_list<int>
    auto A = { 1, 2 };

    // std::initializer_list<int>
    auto B = { 3 };

    // int
    auto C{ 4 };

    // C3535: cannot deduce type for 'auto' from initializer list'
    auto D = { 5, 6.7 };

    // C3518 in a direct-list-initialization context the type for 'auto'
    // can only be deduced from a single initializer expression
    auto E{ 8, 9 };

    return 0;
}

```

Limitações e mensagens de erro

A tabela a seguir lista as restrições sobre o uso da `auto` palavra-chave e a mensagem de erro de diagnóstico correspondente que o compilador emite.

NÚMERO DO ERRO	DESCRIÇÃO
C3530	A <code>auto</code> palavra-chave não pode ser combinada com nenhum outro especificador de tipo.
C3531	Um símbolo declarado com a <code>auto</code> palavra-chave deve ter um inicializador.
C3532	Você usou incorretamente a <code>auto</code> palavra-chave para declarar um tipo. Por exemplo, você declarou um tipo de retorno do método ou matriz.
C3533, C3539	Um parâmetro ou argumento de modelo não pode ser declarado com a <code>auto</code> palavra-chave.
C3535	Um parâmetro de método ou de modelo não pode ser declarado com a <code>auto</code> palavra-chave.
C3536	Um símbolo não pode ser usado antes de ser inicializado. Na prática, isso significa que uma variável não pode ser usada para se inicializar.
C3537	Não é possível converter em um tipo declarado com a <code>auto</code> palavra-chave.
C3538	Todos os símbolos em uma lista de declaradores que são declarados com a <code>auto</code> palavra-chave devem ser resolvidos para o mesmo tipo. Para obter mais informações, consulte declarações e definições .

NÚMERO DO ERRO	DESCRIÇÃO
C3540, C3541	Os operadores <code>sizeof</code> e <code>typeid</code> não podem ser aplicados a um símbolo declarado com a <code>auto</code> palavra-chave.

Exemplos

Esses fragmentos de código ilustram algumas das maneiras pelas quais a `auto` palavra-chave pode ser usada.

As declarações a seguir são equivalentes. Na primeira instrução, a variável `j` é declarada como tipo `int`. Na segunda instrução, a variável `k` é deduzida para ser Type `int` porque a expressão de inicialização (0) é um número inteiro.

```
int j = 0; // Variable j is explicitly type int.
auto k = 0; // Variable k is implicitly type int because 0 is an integer.
```

As seguintes declarações são equivalentes, mas a segunda declaração é mais simples do que a primeira. Um dos motivos mais atraentes para usar a `auto` palavra-chave é a simplicidade.

```
map<int,list<string>>::iterator i = m.begin();
auto i = m.begin();
```

O fragmento de código a seguir declara o tipo de variáveis `iter` e `elem` quando os `for` loops e de intervalo `for` começam.

```
// cl /EHsc /nologo /W4
#include <deque>
using namespace std;

int main()
{
    deque<double> dqDoubleData(10, 0.1);

    for (auto iter = dqDoubleData.begin(); iter != dqDoubleData.end(); ++iter)
    { /* ... */ }

    // prefer range-for loops with the following information in mind
    // (this applies to any range-for with auto, not just deque)

    for (auto elem : dqDoubleData) // COPIES elements, not much better than the previous examples
    { /* ... */ }

    for (auto& elem : dqDoubleData) // observes and/or modifies elements IN-PLACE
    { /* ... */ }

    for (const auto& elem : dqDoubleData) // observes elements IN-PLACE
    { /* ... */ }
}
```

O fragmento de código a seguir usa a `new` declaração de operador e de ponteiro para declarar ponteiros.

```
double x = 12.34;
auto *y = new auto(x), **z = new auto(&x);
```

O próximo fragmento de código declara vários símbolos em cada instrução de declaração. Observe que todos os símbolos em cada instrução são resolvidos para o mesmo tipo.

```
auto x = 1, *y = &x, **z = &y; // Resolves to int.  
auto a(2.01), *b (&a); // Resolves to double.  
auto c = 'a', *d(&c); // Resolves to char.  
auto m = 1, &n = m; // Resolves to int.
```

Esse fragmento de código usa o operador condicional (?:) para declarar a variável `x` como um inteiro com o valor de 200:

```
int v1 = 100, v2 = 200;  
auto x = v1 > v2 ? v1 : v2;
```

O fragmento de código a seguir inicializa `x` a variável para o tipo, a `int` variável `y` para uma referência ao tipo `const int` e a variável `fp` para um ponteiro para uma função que retorna o tipo `int`.

```
int f(int x) { return x; }  
int main()  
{  
    auto x = f(0);  
    const auto& y = f(1);  
    int (*p)(int x);  
    p = f;  
    auto fp = p;  
    //...  
}
```

Confira também

[auto Chaves](#)

[Palavras-chave](#)

[/Zc:auto \(Deduzir tipo de variável\)](#)

[sizeof Operador](#)

[typeid](#)

[operator new](#)

[Declarações e definições](#)

[Exemplos de expressões lambda](#)

[Inicializadores](#)

[decltype](#)

const (C++)

02/09/2020 • 5 minutes to read • [Edit Online](#)

Ao modificar uma declaração de dados, a `const` palavra-chave especifica que o objeto ou a variável não é modificável.

Sintaxe

```
const declaration ;
member-function const ;
```

valores const

A `const` palavra-chave especifica que o valor de uma variável é constante e informa ao compilador para impedir que o programador o modifique.

```
// constant_values1.cpp
int main() {
    const int i = 5;
    i = 10;    // C3892
    i++;      // C2105
}
```

Em C++, você pode usar a `const` palavra-chave em vez da diretiva de pré-processador de `#define` para definir valores constantes. Os valores definidos com `const` estão sujeitos à verificação de tipo e podem ser usados no lugar de expressões constantes. Em C++, você pode especificar o tamanho de uma matriz com uma `const` variável da seguinte maneira:

```
// constant_values2.cpp
// compile with: /c
const int maxarray = 255;
char store_char[maxarray]; // allowed in C++; not allowed in C
```

Em C, os valores constantes usam como padrão o vínculo externo, assim eles podem aparecer somente em arquivos de origem. Em C++, os valores constantes usam como padrão o vínculo interno, que permite que eles apareçam em arquivos de cabeçalho.

A `const` palavra-chave também pode ser usada em declarações de ponteiro.

```
// constant_values3.cpp
int main() {
    char *mybuf = 0, *yourbuf;
    char *const aptr = mybuf;
    *aptr = 'a'; // OK
    aptr = yourbuf; // C3892
}
```

Um ponteiro para uma variável declarada como `const` pode ser atribuído somente a um ponteiro que também é declarado como `const`.

```
// constant_values4.cpp
#include <stdio.h>
int main() {
    const char *mybuf = "test";
    char *yourbuf = "test2";
    printf_s("%s\n", mybuf);

    const char *bptr = mybuf;    // Pointer to constant data
    printf_s("%s\n", bptr);

    // *bptr = 'a';    // Error
}
```

Você pode usar ponteiros para os dados constantes como parâmetros de função para evitar que a função modifique um parâmetro passado por um ponteiro.

Para objetos declarados como `const`, você só pode chamar funções de membro constantes. Isso assegura que o objeto constante nunca seja modificado.

```
birthday.getMonth();    // Okay
birthday.setMonth( 4 ); // Error
```

Você pode chamar funções membro constantes ou não constantes para um objeto não constante. Você também pode sobrepor uma função de membro usando a `const` palavra-chave; isso permite que uma versão diferente da função seja chamada para objetos constantes e não constantes.

Você não pode declarar construtores ou destruidores com a `const` palavra-chave.

funções de membro `const`

A declaração de uma função de membro com a `const` palavra-chave especifica que a função é uma função "somente leitura" que não modifica o objeto para o qual ela é chamada. Uma função de membro de constante não pode modificar nenhum membro de dados não estático nem chamar funções de membros que não sejam constantes. Para declarar uma função de membro constante, coloque a `const` palavra-chave após o parêntese de fechamento da lista de argumentos. A `const` palavra-chave é necessária tanto na declaração quanto na definição.

```

// constant_member_function.cpp
class Date
{
public:
    Date( int mn, int dy, int yr );
    int getMonth() const;      // A read-only function
    void setMonth( int mn );  // A write function; can't be const
private:
    int month;
};

int Date::getMonth() const
{
    return month;           // Doesn't modify anything
}
void Date::setMonth( int mn )
{
    month = mn;            // Modifies data member
}
int main()
{
    Date MyDate( 7, 4, 1998 );
    const Date BirthDate( 1, 18, 1953 );
    MyDate.setMonth( 4 );   // Okay
    BirthDate.getMonth();  // Okay
    BirthDate.setMonth( 4 ); // C2662 Error
}

```

Diferenças entre C e C++ const

Quando você declara uma variável como `const` em um arquivo de código-fonte C, você faz isso da seguinte maneira:

```
const int i = 2;
```

Então, você pode usar essa variável em outro módulo como segue:

```
extern const int i;
```

Mas para obter o mesmo comportamento em C++, você deve declarar sua `const` variável como:

```
extern const int i = 2;
```

Se você quiser declarar uma `extern` variável em um arquivo de código-fonte C++ para uso em um arquivo de código-fonte C, use:

```
extern "C" const int x=10;
```

para evitar a desconfiguração do nome pelo compilador C++.

Comentários

Ao seguir a lista de parâmetros de uma função de membro, a `const` palavra-chave especifica que a função não modifica o objeto para o qual ele é invocado.

Para obter mais informações sobre `const` o, consulte os seguintes tópicos:

- Ponteiros const e voláteis
- Qualificadores de tipo (Referência da linguagem C)
- volatile
- #define

Confira também

Palavras-chave

:::no-loc(constexpr):::C

02/09/2020 • 9 minutes to read • [Edit Online](#)

A palavra-chave `:::no-loc(constexpr):::` foi introduzida no C++ 11 e aprimorada no C++ 14. Significa * `:::no-loc(const):::` expressão Ant*. Como `:::no-loc(const):::`, ele pode ser aplicado a variáveis: um erro do compilador é gerado quando qualquer código tenta obter `:::no-loc(if):::` o valor de `mod y`. Ao contrário `:::no-loc(const):::` de, `:::no-loc(constexpr):::` também pode ser aplicado a funções e à classe `:::no-loc(const):::` ructors.

`:::no-loc(constexpr):::` indica que o valor, ou valor de retorno, é `:::no-loc(const):::` Ant e, quando possível, é calculado no momento da compilação.

Um `:::no-loc(constexpr):::` valor integral pode ser usado sempre `:::no-loc(const):::` que um inteiro é necessário, como em argumentos de modelo e declarações de matriz. E quando um valor é calculado em tempo de compilação em vez de tempo de execução, ele ajuda o programa a ser executado mais rapidamente e usa menos memória.

Para limitar a complexidade de cálculos de Ant em tempo de compilação `:::no-loc(const):::` e seus possíveis impactos no tempo de compilação, o padrão C++ 14 exige que os tipos em `:::no-loc(const):::` expressões de Ant sejam [tipos literais](#).

Sintaxe

```
:::no-loc(constexpr)::: ***tipo literal* ident :::no-loc(if)::: ler = * :::no-loc(const)::: Ant-Expression*
;
:::no-loc(constexpr)::: ***tipo literal* ident :::no-loc(if)::: ler { * :::no-loc(const)::: Ant-Expression* }
;
* :::no-loc(constexpr)::: ***tipo literal ident :::no-loc(if)::: ler( params );
* :::no-loc(constexpr)::: ***ctor( params );
```

parâmetros

`params`

Um ou mais parâmetros, cada um deles deve ser um tipo literal e, por sua vez, deve ser uma `:::no-loc(const):::` expressão Ant.

Valor retornado

Uma `:::no-loc(constexpr):::` variável ou função deve retornar um [tipo literal](#).

Variáveis `:::no-loc(constexpr):::`

A `:::no-loc(if):::` ferência de principal entre `:::no-loc(const):::` e `:::no-loc(constexpr):::` variáveis é que a inicialização de uma `:::no-loc(const):::` variável pode ser adiada até o tempo de execução. Uma `:::no-loc(constexpr):::` variável deve ser inicializada em tempo de compilação. Todas as `:::no-loc(constexpr):::` variáveis são `:::no-loc(const):::`.

- Uma variável pode ser declarada com `:::no-loc(constexpr):::`, quando tem um tipo literal e é inicializada. Se a inicialização for por `:::no-loc(for):::` med por um `:::no-loc(const):::` ructor, o `:::no-loc(const):::` ructor deverá ser declarado como `:::no-loc(constexpr):::`.
- Uma referência pode ser declarada como `:::no-loc(constexpr):::` quando ambas as condições são

atendidas: o objeto referenciado é inicializado por uma `:::no-loc(const)` expressão Ant e todas as conversões implícitas invocadas durante a inicialização também são `:::no-loc(const)` expressões de Ant.

- Todas as declarações de uma `:::no-loc(constexpr)` variável ou função devem ter a `:::no-loc(constexpr)` especificação `:::no-loc(if)` ler.

```
:::no-loc(constexpr)::: float x = 42.0;
:::no-loc(constexpr)::: float y{108};
:::no-loc(constexpr)::: float z = exp(5, 3);
:::no-loc(constexpr)::: int i; // Error! Not initialized
int j = 0;
:::no-loc(constexpr)::: int k = j + 1; //Error! j not a ::::no-loc(const):::ant expression
```

:::no-loc(constexpr):::funções do

Uma `:::no-loc(constexpr)` função é aquela cujo valor de retorno é computáveis no momento da compilação quando o código de consumo exigir. O consumo de código requer o valor de retorno no momento da compilação para inicializar uma `:::no-loc(constexpr)` variável ou para fornecer um argumento de modelo sem tipo.

Quando seus argumentos são `:::no-loc(constexpr)` valores, uma `:::no-loc(constexpr)` função produz um Ant em tempo de compilação `:::no-loc(const)`. Quando chamado com não `:::no-loc(constexpr)` argumentos, ou quando seu valor não é necessário no momento da compilação, ele produz um valor em tempo de execução como uma função regular. (Esse comportamento duplo evita que você precise escrever `:::no-loc(constexpr)` e não `:::no-loc(constexpr)` versões da mesma função.)

Uma `:::no-loc(constexpr)` função ou `:::no-loc(const)` ructor é implicitamente `:::no-loc(inline)`.

As regras a seguir se aplicam a `:::no-loc(constexpr)` funções do:

- Uma `:::no-loc(constexpr)` função deve aceitar e retornar apenas [tipos literais](#).
- Uma `:::no-loc(constexpr)` função pode ser recursiva.
- Não pode ser [virtual](#). Um `:::no-loc(const)` ructor não pode ser definido como `:::no-loc(constexpr)` quando a classe delimitadora tem quaisquer classes base virtuais.
- O corpo pode ser definido como `= default` ou `= delete`.
- O corpo não pode conter `:::no-loc(goto)` instruções ou `:::no-loc(try)` blocos.
- Uma especialização explícita de um não `:::no-loc(constexpr)` modelo pode ser declarada como `:::no-loc(constexpr)`:
- Uma especialização explícita de um `:::no-loc(constexpr)` modelo também não precisa ser `:::no-loc(constexpr)`:

As regras a seguir se aplicam a `:::no-loc(constexpr)` funções no Visual Studio 2017 e posteriores:

- Ele pode conter `:::no-loc(if)` instruções and e `:::no-loc(switch)` todas as instruções de looping, incluindo `:::no-loc(for)` com base em intervalo `:::no-loc(for)`, `:::no-loc(while)` e `do :::no-loc(while)`.
- Ele pode conter declarações de variáveis locais, mas a variável deve ser inicializada. Ele deve ser um tipo literal e não pode ser `static` ou thread-local. A variável declarada localmente não deve ser `:::no-loc(const)` e pode ser mutada.
- Uma `:::no-loc(constexpr)` função não `static` membro não precisa ser implicitamente `:::no-loc(const)`.

```
::::no-loc(constexpr)::: float exp(float x, int n)
{
    return n == 0 ? 1 :
        n % 2 == 0 ? exp(x * x, n / 2) :
        exp(x * x, (n - 1) / 2) * x;
};
```

TIP

No depurador do Visual Studio, ao depurar uma compilação de depuração não otimizada, você pode saber se uma `::::no-loc(constexpr):::` função está sendo avaliada em tempo de compilação, colocando um ponto de interrupção dentro dele. Se o ponto de interrupção for atingido, a função foi chamada em tempo de execução. Caso contrário, a função foi chamada em tempo de compilação.

externo::::no-loc(constexpr):::

A opção de compilador `/Zc: externConstexpr` faz com que o compilador aplique **vínculo externo** a variáveis declaradas usando `**extern ::::no-loc(constexpr)::: **`. Em versões anteriores do Visual Studio, por padrão ou quando `/Zc: externConstexpr-` é `spec ::::no-loc(if):::` negado, o Visual Studio aplica vínculo interno a `::::no-loc(constexpr):::` variáveis, mesmo quando a `extern` palavra-chave é usada. A opção `/Zc: externConstexpr` está disponível a partir da atualização 15,6 do Visual Studio 2017 e está desativada por padrão. A opção `/permissive-` não habilita `/Zc: externConstexpr`.

Exemplo

O exemplo a seguir mostra `::::no-loc(constexpr):::` variáveis, funções e um tipo definido pelo usuário. Na última instrução em `main()`, a `::::no-loc(constexpr):::` função de membro `GetValue()` é uma chamada de tempo de execução porque o valor não precisa ser conhecido no momento da compilação.

```

// ::::no-loc(constexpr):::.cpp
// Compile with: cl /EHsc /W4 ::::no-loc(constexpr):::.cpp
#include <iostream>

using namespace std;

// Pass by value
::::no-loc(constexpr)::: float exp(float x, int n)
{
    return n == 0 ? 1 :
        n % 2 == 0 ? exp(x * x, n / 2) :
        exp(x * x, (n - 1) / 2) * x;
};

// Pass by reference
::::no-loc(constexpr)::: float exp2(::::no-loc(const)::: float& x, ::::no-loc(const)::: int& n)
{
    return n == 0 ? 1 :
        n % 2 == 0 ? exp2(x * x, n / 2) :
        exp2(x * x, (n - 1) / 2) * x;
};

// Compile-time computation of array length
template<typename T, int N>
::::no-loc(constexpr)::: int length(::::no-loc(const)::: T(&)[N])
{
    return N;
}

// Recursive ::::no-loc(constexpr)::: function
::::no-loc(constexpr)::: int fac(int n)
{
    return n == 1 ? 1 : n * fac(n - 1);
}

// User-defined type
class Foo
{
public:
    ::::no-loc(constexpr)::: explicit Foo(int i) : _i(i) {}
    ::::no-loc(constexpr)::: int GetValue() ::::no-loc(const):::
    {
        return _i;
    }
private:
    int _i;
};

int main()
{
    // foo is ::::no-loc(const):::
    ::::no-loc(constexpr)::: Foo foo(5);
    // foo = Foo(6); //Error!

    // Compile time:
    ::::no-loc(constexpr)::: float x = exp(5, 3);
    ::::no-loc(constexpr)::: float y { exp(2, 5) };
    ::::no-loc(constexpr)::: int val = foo.GetValue();
    ::::no-loc(constexpr)::: int f5 = fac(5);
    ::::no-loc(const)::: int nums[] { 1, 2, 3, 4 };
    ::::no-loc(const)::: int nums2[length(nums) * 2] { 1, 2, 3, 4, 5, 6, 7, 8 };

    // Run time:
    cout << "The value of foo is " << foo.GetValue() << endl;
}

```

Requisitos

Visual Studio 2015 ou posterior.

Confira também

[Declarações e definições](#)

`::no-loc(const)::`

:::no-loc(extern):::C

02/09/2020 • 7 minutes to read • [Edit Online](#)

A ::::no-loc(extern)::: palavra-chave pode ser aplicada a uma variável global, função ou declaração de modelo. Ele especifica que o símbolo tem a * ::::no-loc(extern)::: vinculação Al*. Para obter informações básicas sobre vinculação e por que o uso de variáveis globais não é recomendado, consulte [unidades de tradução e vinculação](#).

A ::::no-loc(extern)::: palavra-chave tem quatro significados dependendo do contexto:

- Em uma declaração de ::::no-loc(const)::: variável não global, ::::no-loc(extern)::: especifica que a variável ou função é definida em outra unidade de tradução. O ::::no-loc(extern)::: deve ser aplicado em todos os arquivos, exceto aquele em que a variável é definida.
- Em uma ::::no-loc(const)::: declaração de variável, ele especifica que a variável tem ::::no-loc(extern)::: vínculo al. O ::::no-loc(extern)::: deve ser aplicado a todas as declarações em todos os arquivos. (As ::::no-loc(const)::: variáveis globais têm vínculo interno por padrão.)
- ** ::::no-loc(extern)::: "C"** especifica que a função é definida em outro lugar e usa a Convenção de chamada de linguagem C. O ::::no-loc(extern)::: modificador "C" também pode ser aplicado a várias declarações de função em um bloco.
- Em uma declaração de modelo, ::::no-loc(extern)::: especifica que o modelo já foi instanciado em outro lugar. ::::no-loc(extern)::: informa ao compilador que ele pode reutilizar a outra instância, em vez de criar uma nova no local atual. Para obter mais informações sobre esse uso de ::::no-loc(extern):::, consulte [instanciação explícita](#).

::::no-loc(extern):::vínculo para não ::::no-loc(const)::: globais

Quando o vinculador vê ::::no-loc(extern)::: antes de uma declaração de variável global, ele procura a definição em outra unidade de tradução. As declarações de não ::::no-loc(const)::: variáveis em escopo global são ::::no-loc(extern)::: Al por padrão. Aplica-se somente ::::no-loc(extern)::: às declarações que não fornecem a definição.

```
//fileA.cpp
int i = 42; // declaration and definition

//fileB.cpp
::::no-loc(extern)::: int i; // declaration only. same as i in FileA

//fileC.cpp
::::no-loc(extern)::: int i; // declaration only. same as i in FileA

//fileD.cpp
int i = 43; // LNK2005! 'i' already has a definition.
::::no-loc(extern)::: int i = 43; // same error (::::no-loc(extern)::: is ignored on definitions)
```

::::no-loc(extern):::vínculo para ::::no-loc(const)::: globais

::::no-loc(const)::: Por padrão, uma variável global tem ligação interna. Se você quiser que a variável tenha ::::no-loc(extern)::: vínculo Al, aplique a ::::no-loc(extern)::: palavra-chave à definição e a todas as outras declarações em outros arquivos:

```
//fileA.cpp
:::no-loc(extern)::: ::::no-loc(const)::: int i = 42; // ::::no-loc(extern)::: ::::no-loc(const)::: definition

//fileB.cpp
:::no-loc(extern)::: ::::no-loc(const)::: int i; // declaration only. same as i in FileA
```

:::no-loc(extern)::::::no-loc(constexpr):::vinculação

No Visual Studio 2017 versão 15,3 e anteriores, o compilador sempre deu uma `:::no-loc(constexpr):::` variável de vinculação interna, mesmo quando a variável foi marcada `:::no-loc(extern):::`. No Visual Studio 2017 versão 15,5 e posterior, a opção de compilador `/Zc:::no-loc(extern)::: Constexpr` habilita o comportamento correto de conformidade com padrões. Eventualmente, a opção se tornará o padrão. A `/:::no-loc(permissive):::`-opção não habilita `/Zc:::no-loc(extern)::: Constexpr`.

```
:::no-loc(extern)::: ::::no-loc(constexpr)::: int x = 10; //error LNK2005: "int ::::no-loc(const)::: x" already
defined
```

Se um arquivo de cabeçalho contiver uma variável declarada `:::no-loc(extern)::: ::::no-loc(constexpr):::`, ele deverá ser marcado `__declspec(selectany)` corretamente para ter suas declarações duplicadas combinadas:

```
:::no-loc(extern)::: ::::no-loc(constexpr)::: __declspec(selectany) int x = 10;
```

:::no-loc(extern):::Declarações de função "C" e :::no-loc(extern)::: "C++"

Em C++, quando usado com uma cadeia de caracteres, `:::no-loc(extern):::` especifica que as convenções de vinculação de outro idioma estão sendo usadas para o declarador (es). As funções e os dados de C podem ser acessados somente se eles forem declarados anteriormente como tendo vínculo C. No entanto, devem ser definidos em uma unidade de conversão compilada separadamente.

O Microsoft C++ dá suporte às cadeias de caracteres "C" e "C++" no campo de *cadeia literal*. Todos os arquivos de inclusão padrão usam a sintaxe `** :::no-loc(extern)::: "C"**` para permitir que as funções de biblioteca de tempo de execução sejam usadas em programas C++.

Exemplo

O exemplo a seguir mostra como declarar nomes que têm vínculo C:

```

// Declare printf with C linkage.
:::no-loc(extern)::: "C" int printf(:::no-loc(const)::: char *fmt, ...);

// Cause everything in the specified
// header files to have C linkage.
:::no-loc(extern)::: "C" {
    // add your #include statements here
#include <stdio.h>
}

// Declare the two functions ShowChar
// and GetChar with C linkage.
:::no-loc(extern)::: "C" {
    char ShowChar(char ch);
    char GetChar(void);
}

// Define the two functions
// ShowChar and GetChar with C linkage.
:::no-loc(extern)::: "C" char ShowChar(char ch) {
    putchar(ch);
    return ch;
}

:::no-loc(extern)::: "C" char GetChar(void) {
    char ch;
    ch = getchar();
    return ch;
}

// Declare a global variable, errno, with C linkage.
:::no-loc(extern)::: "C" int errno;

```

Se uma função tiver mais de uma especificação de vinculação, ela deverá concordar. É um erro declarar funções como tendo vínculos C e C++. Além disso, se duas declarações para uma função ocorrem em um programa — uma com uma especificação de vinculação e a outra sem — a declaração com a especificação de vinculação deve ser a primeira. Todas as declarações redundantes de funções que já têm a especificação de vinculação são atribuídas a uma vinculação especificada na primeira declaração. Por exemplo:

```

:::no-loc(extern)::: "C" int CFunc1();
...
int CFunc1();           // Redefinition is benign; C linkage is
                        // retained.

int CFunc2();
...
:::no-loc(extern)::: "C" int CFunc2(); // Error: not the first declaration of
                        // CFunc2;  cannot contain linkage
                        // specifier.

```

Confira também

[Palavras-chave](#)

[Unidades de tradução e vinculação](#)

[:::no-loc\(extern\)::: Especificador de classe de armazenamento em C](#)

[Comportamento de identificadores em C](#)

[Ligaçāo em C](#)

Inicializadores

02/09/2020 • 20 minutes to read • [Edit Online](#)

Um inicializador especifica o valor inicial de uma variável. Você pode inicializar variáveis nestes contextos:

- Na definição de uma variável:

```
int i = 3;
Point p1{ 1, 2 };
```

- Como um dos parâmetros de uma função:

```
set_point(Point{ 5, 6 });
```

- Como o valor de retorno de uma função:

```
Point get_new_point(int x, int y) { return { x, y }; }
Point get_new_point(int x, int y) { return Point{ x, y }; }
```

Os inicializadores podem usar estes formatos:

- Uma expressão (ou uma lista de expressões separadas por vírgulas) entre parênteses:

```
Point p1(1, 2);
```

- Um sinal de igual seguido por uma expressão:

```
string s = "hello";
```

- Uma lista de inicializadores entre chaves. A lista pode estar vazia ou pode consistir em um conjunto de listas, como no exemplo a seguir:

```
struct Point{
    int x;
    int y;
};

class PointConsumer{
public:
    void set_point(Point p){};
    void set_points(initializer_list<Point> my_list){};
};

int main() {
    PointConsumer pc{};
    pc.set_point({}); // empty list
    pc.set_point({ 3, 4 });
    pc.set_points({ { 3, 4 }, { 5, 6 } });
}
```

Tipos de inicialização

Há vários tipos de inicialização que podem ocorrer em pontos diferentes na execução do programa. Os tipos diferentes de inicialização não são mutuamente exclusivos — por exemplo, a inicialização de lista pode acionar a inicialização de valor e, em outras circunstâncias, pode acionar a inicialização de agregação.

Inicialização zero

A inicialização do zero é a configuração de uma variável para um valor zero convertido implicitamente no tipo:

- As variáveis numéricas são inicializadas como 0 (ou 0,0 ou 0,0000000000, etc.).
- As variáveis Char são inicializadas para `'\0'`.
- Os ponteiros são inicializados para `nullptr`.
- Matrizes, classes de [Pod](#), structs e uniões têm seus membros inicializados para um valor zero.

A inicialização do zero é executada em diferentes momentos:

- Na inicialização do programa, para todas as variáveis nomeadas que têm a duração estática. Essas variáveis podem ser inicializadas novamente mais tarde.
- Durante a inicialização de valor, para tipos escalares e tipos de classe POD que são inicializados usando chaves vazias.
- Para matrizes que têm apenas um subconjunto de seus membros inicializado.

Veja alguns exemplos de inicialização do zero:

```
struct my_struct{
    int i;
    char c;
};

int i0;           // zero-initialized to 0
int main() {
    static float f1; // zero-initialized to 0.00000000
    double d{};     // zero-initialized to 0.0000000000000000
    int* ptr{};     // initialized to nullptr
    char s_array[3]{'a', 'b'}; // the third char is initialized to '\0'
    int int_array[5] = { 8, 9, 10 }; // the fourth and fifth ints are initialized to 0
    my_struct a_struct{}; // i = 0, c = '\0'
}
```

Inicialização padrão

A inicialização padrão para classes, structs e uniões é a inicialização com um construtor padrão. O construtor padrão pode ser chamado sem nenhuma expressão de inicialização ou com a `new` palavra-chave:

```
MyClass mc1;
MyClass* mc3 = new MyClass;
```

Se a classe, estrutura ou união não tiver um construtor diferente, o compilador emitirá um erro.

As variáveis escalares são inicializadas por padrão quando são definidas sem nenhuma expressão de inicialização. Elas têm valores indeterminados.

```
int i1;
float f;
char c;
```

As matrizes são inicializadas por padrão quando são definidas sem nenhuma expressão de inicialização. Quando

uma matriz é inicializada por padrão, seus membros são inicializados por padrão e têm valores indeterminados, como no exemplo a seguir:

```
int int_arr[3];
```

Se os membros da matriz não tiverem um construtor padrão, o compilador emitirá um erro.

Inicialização padrão de variáveis constantes

As variáveis constantes devem ser declaradas juntamente com um inicializador. Se forem tipos escalares, eles causarão um erro de compilador e, se forem tipos de classe que têm um construtor padrão, causarão um aviso:

```
class MyClass{};  
int main() {  
    //const int i2;    // compiler error C2734: const object must be initialized if not extern  
    //const char c2;  // same error  
    const MyClass mc1; // compiler error C4269: 'const automatic data initialized with compiler generated  
    default constructor produces unreliable results  
}
```

Inicialização padrão de variáveis estáticas

Variáveis estáticas declaradas com nenhum inicializador são inicializadas como 0 (implicitamente convertida para o tipo).

```
class MyClass {  
private:  
    int m_int;  
    char m_char;  
};  
  
int main() {  
    static int int1;      // 0  
    static char char1;   // '\0'  
    static bool bool1;   // false  
    static MyClass mc1;  // {0, '\0'}  
}
```

Para obter mais informações sobre a inicialização de objetos estáticos globais, consulte [principal função e argumentos de linha de comando](#).

Inicialização de valor

A inicialização do valor ocorre nos seguintes casos:

- um valor nomeado é inicializado usando a inicialização de chave vazia
- um objeto temporário anônimo é inicializado usando parênteses ou chaves vazias
- um objeto é inicializado com a `new` palavra-chave mais parênteses vazios ou chaves

A inicialização do valor faz o seguinte:

- para classes com pelo menos um construtor público, o construtor padrão é chamado
- para classes não Union sem construtores declarados, o objeto é inicializado com zero e o construtor padrão é chamado
- para matrizes, cada elemento é inicializado por valor
- em todos os outros casos, a variável é zero inicializada

```

class BaseClass {
private:
    int m_int;
};

int main() {
    BaseClass bc{};      // class is initialized
    BaseClass* bc2 = new BaseClass(); // class is initialized, m_int value is 0
    int int_arr[3]{0}; // value of all members is 0
    int a{0};          // value of a is 0
    double b{0};        // value of b is 0.0000000000000000
}

```

Inicialização da cópia

A inicialização da cópia é a inicialização de um objeto usando um objeto diferente. Isso ocorre nos seguintes casos:

- uma variável é inicializada usando um sinal de igual
- um argumento é passado para uma função
- um objeto é retornado de uma função
- uma exceção é lançada ou detectada
- um membro de dados não estático é inicializado usando um sinal de igual
- Membros de classe, struct e Union são inicializados pela inicialização de cópia durante a inicialização de agregação. Consulte [inicialização de agregação](#) para obter exemplos.

O código a seguir mostra vários exemplos de inicialização de cópia:

```

#include <iostream>
using namespace std;

class MyClass{
public:
    MyClass(int myInt) {}
    void set_int(int myInt) { m_int = myInt; }
    int get_int() const { return m_int; }
private:
    int m_int = 7; // copy initialization of m_int

};

class MyException : public exception{};

int main() {
    int i = 5;           // copy initialization of i
    MyClass mc1{ i };
    MyClass mc2 = mc1;    // copy initialization of mc2 from mc1
    MyClass mc1.set_int(i); // copy initialization of parameter from i
    int i2 = mc2.get_int(); // copy initialization of i2 from return value of get_int()

    try{
        throw MyException();
    }
    catch (MyException ex){ // copy initialization of ex
        cout << ex.what();
    }
}

```

A inicialização da cópia não pode invocar construtores explícitos.

```
vector<int> v = 10; // the constructor is explicit; compiler error C2440: cannot convert from 'int' to
'std::vector<int, std::allocator<_Ty>>'
regex r = "a.*b"; // the constructor is explicit; same error
shared_ptr<int> sp = new int(1729); // the constructor is explicit; same error
```

Em alguns casos, se o construtor de cópia da classe for excluído ou estiver inacessível, a inicialização de cópia emitirá um erro de compilador.

Inicialização direta

A inicialização direta é a inicialização usando chaves (não vazias) ou parênteses. Diferentemente da inicialização de cópia, ela pode invocar construtores explícitos. Isso ocorre nos seguintes casos:

- uma variável foi inicializada com chaves não vazias ou parênteses
- uma variável é inicializada com a `new` palavra-chave mais chaves não vazias ou parênteses
- uma variável é inicializada com `** static_cast **`
- em um construtor, as classes base e os membros não estáticos são inicializados com uma lista de inicializadores
- na cópia de uma variável capturada dentro de uma expressão lambda

O código a seguir mostra alguns exemplos de inicialização direta:

```
class BaseClass{
public:
    BaseClass(int n) :m_int(n){} // m_int is direct initialized
private:
    int m_int;
};

class DerivedClass : public BaseClass{
public:
    // BaseClass and m_char are direct initialized
    DerivedClass(int n, char c) : BaseClass(n), m_char(c) {}
private:
    char m_char;
};

int main(){
    BaseClass bc1(5);
    DerivedClass dc1{ 1, 'c' };
    BaseClass* bc2 = new BaseClass(7);
    BaseClass bc3 = static_cast<BaseClass>(dc1);

    int a = 1;
    function<int()> func = [a](){ return a + 1; }; // a is direct initialized
    int n = func();
}
```

Inicialização de lista

A inicialização da lista ocorre quando uma variável é inicializada usando uma lista de inicializadores de chaves. Listas de inicializadores de chaves podem ser usadas nos seguintes casos:

- uma variável foi inicializada
- uma classe é inicializada com a `new` palavra-chave
- um objeto é retornado de uma função
- um argumento passado para uma função

- um dos argumentos em uma inicialização direta
- em um inicializador de membro de dados não estático
- em uma lista de inicializadores de Construtor

O código a seguir mostra alguns exemplos de inicialização de lista:

```

class MyClass {
public:
    MyClass(int myInt, char myChar) {}
private:
    int m_int[]{ 3 };
    char m_char;
};

class MyClassConsumer{
public:
    void set_class(MyClass c) {}
    MyClass get_class() { return MyClass{ 0, '\0' }; }
};

struct MyStruct{
    int my_int;
    char my_char;
    MyClass my_class;
};

int main() {
    MyClass mc1{ 1, 'a' };
    MyClass* mc2 = new MyClass{ 2, 'b' };
    MyClass mc3 = { 3, 'c' };

    MyClassConsumer mcc;
    mcc.set_class(MyClass{ 3, 'c' });
    mcc.set_class({ 4, 'd' });

    MyStruct ms1{ 1, 'a', { 2, 'b' } };
}

```

Inicialização de agregação

A inicialização de agregação é uma forma da inicialização de lista para tipos de classe ou matrizes (muitas vezes, uniões ou estruturas) que não tenham:

- nenhum membro privado ou protegido
- Não há construtores fornecidos pelo usuário, exceto para construtores explicitamente padronizados ou excluídos
- nenhuma classe base
- nenhuma função de membro virtual

NOTE

No Visual Studio 2015 e anterior, uma agregação não tem permissão para ter inicializadores de chave ou igual para membros não estáticos. Essa restrição foi removida no padrão C++ 14 e implementada no Visual Studio 2017.

Inicializadores agregados consistem em uma lista de inicialização entre chaves, com ou sem um sinal de igual, como no exemplo a seguir:

```

#include <iostream>
using namespace std;

struct MyAggregate{
    int myInt;
    char myChar;
};

struct MyAggregate2{
    int myInt;
    char myChar = 'Z'; // member-initializer OK in C++14
};

int main() {
    MyAggregate agg1{ 1, 'c' };
    MyAggregate2 agg2{2};
    cout << "agg1: " << agg1.myChar << ":" << agg1.myInt << endl;
    cout << "agg2: " << agg2.myChar << ":" << agg2.myInt << endl;

    int myArr1[] = { 1, 2, 3, 4 };
    int myArr2[3] = { 5, 6, 7 };
    int myArr3[5] = { 8, 9, 10 };

    cout << "myArr1: ";
    for (int i : myArr1){
        cout << i << " ";
    }
    cout << endl;

    cout << "myArr3: ";
    for (auto const &i : myArr3) {
        cout << i << " ";
    }
    cout << endl;
}

```

Você deve ver o seguinte resultado:

```

agg1: c: 1
agg2: Z: 2
myArr1: 1 2 3 4
myArr3: 8 9 10 0 0

```

IMPORTANT

Membros de matriz que são declarados, mas não explicitamente inicializados durante a inicialização de agregação, são inicializados com zero, como no `myArr3` acima.

Inicializando uniões e structs

Se uma União não tiver um construtor, você poderá inicializá-lo com um único valor (ou com outra instância de uma União). O valor é usado para inicializar o primeiro campo não estático. Essa inicialização é diferente da inicialização de estrutura, na qual o primeiro valor no inicializador é usado para inicializar o primeiro campo, o segundo valor para inicializar o segundo campo e assim por diante. Compare a inicialização de uniões e structs no exemplo a seguir:

```

struct MyStruct {
    int myInt;
    char myChar;
};

union MyUnion {
    int my_int;
    char my_char;
    bool my_bool;
    MyStruct my_struct;
};

int main() {
    MyUnion mu1{ 'a' }; // my_int = 97, my_char = 'a', my_bool = true, {myInt = 97, myChar = '\0'}
    MyUnion mu2{ 1 }; // my_int = 1, my_char = 'x1', my_bool = true, {myInt = 1, myChar = '\0'}
    MyUnion mu3{}; // my_int = 0, my_char = '\0', my_bool = false, {myInt = 0, myChar = '\0'}
    MyUnion mu4 = mu3; // my_int = 0, my_char = '\0', my_bool = false, {myInt = 0, myChar = '\0'}
    //MyUnion mu5{ 1, 'a', true }; // compiler error: C2078: too many initializers
    //MyUnion mu6 = 'a'; // compiler error: C2440: cannot convert from 'char' to 'MyUnion'
    //MyUnion mu7 = 1; // compiler error: C2440: cannot convert from 'int' to 'MyUnion'

    MyStruct ms1{ 'a' }; // myInt = 97, myChar = '\0'
    MyStruct ms2{ 1 }; // myInt = 1, myChar = '\0'
    MyStruct ms3{}; // myInt = 0, myChar = '\0'
    MyStruct ms4{1, 'a'}; // myInt = 1, myChar = 'a'
    MyStruct ms5 = { 2, 'b' }; // myInt = 2, myChar = 'b'
}

```

Inicializando agregações que contêm agregações

Os tipos de agregação podem conter outros tipos de agregação, por exemplo matrizes de matrizes, matrizes de structs e assim por diante. Esses tipos são inicializados usando conjuntos aninhados de chaves, por exemplo:

```

struct MyStruct {
    int myInt;
    char myChar;
};

int main() {
    int intArr1[2][2]{{ 1, 2 }, { 3, 4 }};
    int intArr3[2][2] = {1, 2, 3, 4};
    MyStruct structArr[]{{ 1, 'a' }, { 2, 'b' }, {3, 'c'} };
}

```

Inicialização de referência

Variáveis do tipo de referência devem ser inicializadas com um objeto do tipo do qual o tipo de referência é derivado, ou com um objeto de um tipo que possa ser convertido para o tipo do qual o tipo de referência é derivado. Por exemplo:

```

// initializing_references.cpp
int iVar;
long lVar;
int main()
{
    long& LongRef1 = lVar; // No conversion required.
    long& LongRef2 = iVar; // Error C2440
    const long& LongRef3 = iVar; // OK
    LongRef1 = 23L; // Change lVar through a reference.
    LongRef2 = 11L; // Change iVar through a reference.
    LongRef3 = 11L; // Error C3892
}

```

A única maneira de inicializar uma referência com um objeto temporário é inicializar um objeto temporário constante. Após a inicialização, uma variável de tipo de referência sempre aponta para o mesmo objeto; ela não

pode ser modificada para apontar para outro.

Embora a sintaxe possa ser a mesma, a inicialização das variáveis do tipo de referência e atribuição para as variáveis do tipo de referência são semanticamente diferentes. No exemplo anterior, as atribuições que modificam `iVar` e `lVar` são semelhantes às inicializações, mas têm efeitos diferentes. A inicialização especifica o objeto para o qual os pontos das variáveis do tipo de referência apontam; a atribuição atribui o objeto mencionado pela referência.

Como passam um argumento do tipo de referência para uma função e retornam um valor do tipo de referência de uma função que são inicializações, os argumentos formais para uma função são inicializados corretamente e as referências são retornadas.

As variáveis do tipo de referência podem ser declaradas sem inicializadores, apenas na seguinte maneira:

- Declarações de função (protótipos). Por exemplo:

```
int func( int& );
```

- Declarações de tipo de retorno de função. Por exemplo:

```
int& func( int& );
```

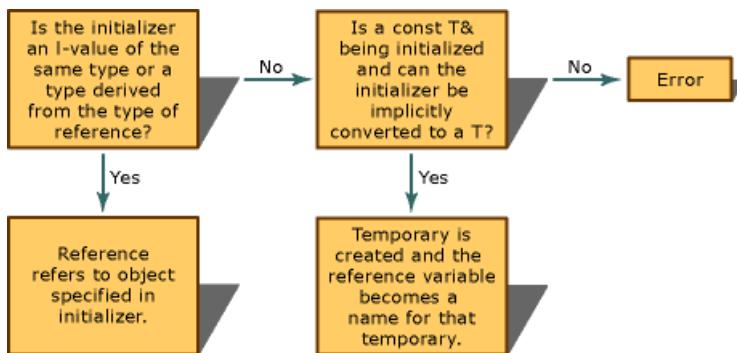
- Declaração de um membro da classe do tipo referência. Por exemplo:

```
class c {public:    int& i;};
```

- Declaração de uma variável explicitamente especificada como `extern`. Por exemplo:

```
extern int& iVal;
```

Ao inicializar uma variável do tipo de referência, o compilador usa o gráfico de decisão mostrado na figura a seguir para selecionar entre a criação de uma referência para um objeto ou a criação de um objeto temporário para o qual a referência aponta.



Grafo de decisão para inicialização de tipos de referência

Referências a `volatile` tipos (declarados como identificador de `volatile TypeName& identifier`) podem ser inicializadas com `volatile` objetos do mesmo tipo ou com objetos que não foram declarados como `volatile`. No entanto, eles não podem ser inicializados com `const` objetos desse tipo. Da mesma forma, referências a `const` tipos (declarados como identificador de `const TypeName& identifier`) podem ser inicializadas com `const` objetos do mesmo tipo (ou qualquer coisa que tenha uma conversão para esse tipo ou com objetos que não tenham sido declarados como `const`). No entanto, eles não podem ser inicializados com `volatile` objetos desse tipo.

As referências que não são qualificadas com `const` a `volatile` palavra-chave ou podem ser inicializadas somente com objetos declarados como nem `const` nem `volatile` .

Inicialização de variáveis externas

As declarações de variáveis automáticas, estáticas e externas podem conter inicializadores. No entanto, as declarações de variáveis externas podem conter inicializadores somente se as variáveis não forem declaradas como `extern` .

Aliases e typedefs (C++)

02/09/2020 • 11 minutes to read • [Edit Online](#)

Você pode usar uma *declaração de alias* para declarar um nome a ser usado como um sinônimo para um tipo anteriormente declarado. (Esse mecanismo também é referenciado informalmente como um *alias de tipo*). Você também pode usar esse mecanismo para criar um *modelo de alias*, que pode ser particularmente útil para alocadores personalizados.

Sintaxe

```
using identifier = type;
```

Comentários

ID

O nome do alias.

tipo

O identificador de tipo para o qual você está criando um alias.

Um alias não introduz um novo tipo e não podem alterar o significado de um nome de tipo existente.

A forma mais simples de um alias é equivalente ao `typedef` mecanismo do C++ 03:

```
// C++11
using counter = long;

// C++03 equivalent:
// typedef long counter;
```

Os dois permitem a criação de variáveis do tipo "counter". Algo mais útil seria um alias de tipo para

`std::ios_base::fmtflags` como este:

```
// C++11
using fmtfl = std::ios_base::fmtflags;

// C++03 equivalent:
// typedef std::ios_base::fmtflags fmtfl;

fmtfl fl_orig = std::cout.flags();
fmtfl fl_hex = (fl_orig & ~std::cout.basefield) | std::cout.showbase | std::cout.hex;
// ...
std::cout.flags(fl_hex);
```

Os aliases também funcionam com ponteiros de função, mas são muito mais legíveis do que o `typedef` equivalente:

```
// C++11
using func = void(*)(int);

// C++03 equivalent:
// typedef void (*func)(int);

// func can be assigned to a function pointer value
void actual_function(int arg) { /* some code */ }
func fptr = &actual_function;
```

Uma limitação do `typedef` mecanismo é que ele não funciona com modelos. No entanto, a sintaxe de alias de tipo no C++11 permite a criação de modelos de alias:

```
template<typename T> using ptr = T*;

// the name 'ptr<T>' is now an alias for pointer to T
ptr<int> ptr_int;
```

Exemplo

O exemplo a seguir demonstra como usar um modelo de alias com um alocador personalizado – nesse caso, um tipo de vetor de inteiro. Você pode substituir qualquer tipo para `int` para criar um alias conveniente para ocultar as listas de parâmetros complexos em seu código funcional principal. Ao usar o alocador personalizado em todo o seu código, você pode melhorar a legibilidade e reduzir o risco de introduzir bugs causados por erros de digitação.

```

#include <stdlib.h>
#include <new>

template <typename T> struct MyAlloc {
    typedef T value_type;

    MyAlloc() { }
    template <typename U> MyAlloc(const MyAlloc<U>&) { }

    bool operator==(const MyAlloc&) const { return true; }
    bool operator!=(const MyAlloc&) const { return false; }

    T * allocate(const size_t n) const {
        if (n == 0) {
            return nullptr;
        }

        if (n > static_cast<size_t>(-1) / sizeof(T)) {
            throw std::bad_array_new_length();
        }

        void * const pv = malloc(n * sizeof(T));

        if (!pv) {
            throw std::bad_alloc();
        }

        return static_cast<T *>(pv);
    }

    void deallocate(T * const p, size_t) const {
        free(p);
    }
};

#include <vector>
using MyIntVector = std::vector<int, MyAlloc<int>>;

#include <iostream>

int main ()
{
    MyIntVector foov = { 1701, 1764, 1664 };

    for (auto a: foov) std::cout << a << " ";
    std::cout << "\n";

    return 0;
}

```

```
1701 1764 1664
```

TypeDefs

Uma `typedef` declaração apresenta um nome que, dentro de seu escopo, se torna um sinônimo para o tipo fornecido pela parte da *declaração de tipo* da declaração.

Você pode usar declarações de `typedef` para construir nomes mais curtos ou mais significativos para os tipos já definidos pelo idioma ou para os tipos que você declarou. Os nomes de `typedef` permitem que você encapsule os detalhes da implementação que podem ser alterados.

Em contraste com as `class`, `struct`, `union` e `enum`, as `typedef` declarações não introduzem

novos tipos — elas introduzem novos nomes para os tipos existentes.

Nomes declarados usando `typedef` ocupam o mesmo namespace que outros identificadores (exceto rótulos de instrução). Portanto, não podem usar o mesmo identificador de um nome declarado anteriormente, exceto em uma declaração de tipo de classe. Considere o exemplo a seguir:

```
// typedef_names1.cpp
// C2377 expected
typedef unsigned long UL;    // Declare a typedef name, UL.
int UL;                      // C2377: redefined.
```

As regras que ocultam o nome que pertencem a outros identificadores também regem a visibilidade dos nomes declarados usando `typedef`. Portanto, o exemplo a seguir é válido em C++:

```
// typedef_names2.cpp
typedef unsigned long UL;    // Declare a typedef name, UL
int main()
{
    unsigned int UL;    // Redefinition hides typedef name
}

// typedef UL back in scope
```

```
// typedef_specifier1.cpp
typedef char FlagType;

int main()
{
}

void myproc( int )
{
    int FlagType;
}
```

Ao declarar um identificador de escopo local com o mesmo nome de um `typedef`, ou ao declarar um membro de uma estrutura ou de uma união no mesmo escopo ou em um escopo interno, o especificador do tipo deverá ser especificado. Por exemplo:

```
typedef char FlagType;
const FlagType x;
```

Para reutilizar o nome `FlagType` para um identificador, um membro da estrutura ou da união, o tipo deve ser fornecido:

```
const int FlagType; // Type specifier required
```

Não é suficiente indicar

```
const FlagType; // Incomplete specification
```

pois `FlagType` é tomado como parte do tipo, não como um identificador que redeclarado. Esta declaração é executada para ser uma declaração inválida como

```
int; // Illegal declaration
```

Você pode declarar qualquer tipo com `typedef`, incluindo o ponteiro, função e tipos de matriz. Você pode declarar um nome de `typedef` para um ponteiro para um tipo de estrutura ou união antes de definir o tipo da estrutura ou de união, contanto que a definição tenha a mesma visibilidade da declaração.

Exemplos

Um uso de `typedef` declarações é tornar as declarações mais uniformes e compactas. Por exemplo:

```
typedef char CHAR;           // Character type.
typedef CHAR * PSTR;         // Pointer to a string (char *).
PSTR strchr( PSTR source, CHAR target );
typedef unsigned long ulong;
ulong ul; // Equivalent to "unsigned long ul;"
```

Para usar `typedef` o para especificar tipos fundamentais e derivados na mesma declaração, você pode separar declaradores com vírgulas. Por exemplo:

```
typedef char CHAR, *PSTR;
```

O exemplo a seguir fornece o tipo `DRAWF` para uma função que não retorna valor e que usa dois argumentos `int`:

```
typedef void DRAWF( int, int );
```

Após a `typedef` instrução acima, a declaração

```
DRAWF box;
```

é equivalente à declaração

```
void box( int, int );
```

`typedef` geralmente é combinado com `struct` para declarar e nomear tipos definidos pelo usuário:

```
// typedefSpecifier2.cpp
#include <stdio.h>

typedef struct mystructtag
{
    int    i;
    double f;
} mystruct;

int main()
{
    mystruct ms;
    ms.i = 10;
    ms.f = 0.99;
    printf_s("%d  %f\n", ms.i, ms.f);
}
```

```
10  0.990000
```

Redeclaração de TYPEDEFs

A `typedef` declaração pode ser usada para redeclarar o mesmo nome para fazer referência ao mesmo tipo. Por exemplo:

```
// FILE1.H
typedef char CHAR;

// FILE2.H
typedef char CHAR;

// PROG.CPP
#include "file1.h"
#include "file2.h" // OK
```

O programa *PROG.CPP* inclui dois arquivos de cabeçalho, ambos os quais contêm `typedef` declarações para o nome `CHAR`. Contanto que as duas declarações façam referência ao mesmo tipo, essa redeclaração é aceitável.

Um `typedef` não pode redefinir um nome que foi declarado anteriormente como um tipo diferente. Portanto, se *arquivo2.H* contém

```
// FILE2.H
typedef int CHAR; // Error
```

o compilador emite um erro devido à tentativa de redeclarar o nome `CHAR` para fazer referência a um tipo diferente. Isso se estende a construtos como:

```
typedef char CHAR;
typedef CHAR CHAR; // OK: redeclared as same type

typedef union REGS // OK: name REGS redeclared
{
    // by typedef name with the
    struct wordregs x; // same meaning.
    struct byteregs h;
} REGS;
```

TYPEDEFs em C++ VS. C

O uso do `typedef` especificador com tipos de classe tem suporte em grande parte devido à prática ANSI C de declarar estruturas não nomeadas em `typedef` declarações. Por exemplo, muitos desenvolvedores de C usam o seguinte:

```
// typedef_with_class_types1.cpp
// compile with: /c
typedef struct { // Declare an unnamed structure and give it the
    // typedef name POINT.
    unsigned x;
    unsigned y;
} POINT;
```

A vantagem dessa declaração é que ela permite declarações como:

```
POINT ptOrigin;
```

em vez de:

```
struct point_t ptOrigin;
```

Em C++, a diferença entre os `typedef` nomes e os tipos reais (declarados com as `class` `struct` palavras-chave, e `union` `enum`) é mais distinta. Embora a prática C de declarar uma estrutura sem nome em uma `typedef` instrução ainda funcione, ela não fornece nenhum benefício de notação como acontece em C.

```
// typedef_with_class_types2.cpp
// compile with: /c /W1
typedef struct {
    int POINT();
    unsigned x;
    unsigned y;
} POINT;
```

O exemplo anterior declara uma classe chamada `POINT` usando a sintaxe de classe sem nome `typedef`. `POINT` é tratado como um nome de classe; no entanto, as seguintes restrições se aplicam a nomes apresentados dessa forma:

- O nome (o sinônimo) não pode aparecer após `class` um `struct` prefixo, ou `union`.
- O nome não pode ser usado como nomes de construtor ou destruidor dentro de uma declaração da classe.

Em resumo, essa sintaxe não fornece um mecanismo para herança, construção ou destruição.

usando declaração

02/09/2020 • 8 minutes to read • [Edit Online](#)

A `using` declaração apresenta um nome para a região declarativa na qual a declaração `using` é exibida.

Sintaxe

```
using [typename] nested-name-specifier unqualified-id ;  
using declarator-list ;
```

parâmetros

especificador de nome aninhado Uma sequência de nomes de namespace, classe ou enumeração e operadores de resolução de escopo (::), terminados por um operador de resolução de escopo. Um operador de resolução de escopo único pode ser usado para introduzir um nome do namespace global. A palavra-chave `typename` é opcional e pode ser usada para resolver nomes dependentes quando introduzido em um modelo de classe de uma classe base.

ID não qualificada Uma expressão de ID não qualificada, que pode ser um identificador, um nome de operador sobreescarregado, um operador literal definido pelo usuário ou um nome de função de conversão, um nome de destruidor de classe ou um nome de modelo e uma lista de argumentos.

Declarador-lista Uma lista separada por vírgulas de [`typename`] declaradores de *ID não qualificados* do *especificador de nome aninhado*, seguido opcionalmente por uma elipse.

Comentários

Uma declaração `using` introduz um nome não qualificado como um sinônimo para uma entidade declarada em outro lugar. Ele permite que um único nome de um namespace específico seja usado sem qualificação explícita na região de declaração na qual ele aparece. Isso é diferente da [diretiva using](#), que permite que *todos* os nomes em um namespace sejam usados sem qualificação. A `using` palavra-chave também é usada para [aliases de tipo](#).

Exemplo

Uma declaração `using` pode ser usada em uma definição de classe.

```

// using_declaration1.cpp
#include <stdio.h>
class B {
public:
    void f(char) {
        printf_s("In B::f()\n");
    }

    void g(char) {
        printf_s("In B::g()\n");
    }
};

class D : B {
public:
    using B::f;    // B::f(char) is now visible as D::f(char)
    using B::g;    // B::g(char) is now visible as D::g(char)
    void f(int) {
        printf_s("In D::f()\n");
        f('c');    // Invokes B::f(char) instead of recursing
    }

    void g(int) {
        printf_s("In D::g()\n");
        g('c');    // Invokes B::g(char) instead of recursing
    }
};

int main() {
    D myD;
    myD.f(1);
    myD.g('a');
}

```

```

In D::f()
In B::f()
In B::g()

```

Exemplo

Quando usada para declarar um membro, uma declaração using deve se referir a um membro de uma classe base.

```
// using_declaration2.cpp
#include <stdio.h>

class B {
public:
    void f(char) {
        printf_s("In B::f()\n");
    }

    void g(char) {
        printf_s("In B::g()\n");
    }
};

class C {
public:
    int g();
};

class D2 : public B {
public:
    using B::f;    // ok: B is a base of D2
    // using C::g;    // error: C isn't a base of D2
};

int main() {
    D2 MyD2;
    MyD2.f('a');
}
```

```
In B::f()
```

Exemplo

Os membros declarados usando uma declaração using podem ser referenciados usando a qualificação explícita. O prefixo `::` refere-se ao namespace global.

```

// using_declaration3.cpp
#include <stdio.h>

void f() {
    printf_s("In f\n");
}

namespace A {
    void g() {
        printf_s("In A::g\n");
    }
}

namespace X {
    using ::f;    // global f is also visible as X::f
    using A::g;   // A's g is now visible as X::g
}

void h() {
    printf_s("In h\n");
    X::f();      // calls ::f
    X::g();      // calls A::g
}

int main() {
    h();
}

```

```

In h
In f
In A::g

```

Exemplo

Quando uma declaração `using` é feita, o sinônimo criado pela declaração se refere apenas às definições que são válidas no ponto da declaração `using`. As definições adicionadas a um namespace depois da declaração `using` são sinônimos inválidos.

Um nome definido por uma `using` declaração é um alias para seu nome original. Não afeta o tipo, a vinculação ou outros atributos da declaração original.

```

// post_declaration_namespace_additions.cpp
// compile with: /c
namespace A {
    void f(int) {}
}

using A::f;    // f is a synonym for A::f(int) only

namespace A {
    void f(char) {}
}

void f() {
    f('a');    // refers to A::f(int), even though A::f(char) exists
}

void b() {
    using A::f; // refers to A::f(int) AND A::f(char)
    f('a');    // calls A::f(char);
}

```

Exemplo

Com relação às funções nos namespaces, se um conjunto de declarações locais e declarações using para um único nome for fornecido em uma região declarativa, todas as declarações deverão referenciar a mesma entidade, ou referenciar funções.

```
// functions_in_namespaces1.cpp
// C2874 expected
namespace B {
    int i;
    void f(int);
    void f(double);
}

void g() {
    int i;
    using B::i;    // error: i declared twice
    void f(char);
    using B::f;    // ok: each f is a function
}
```

No exemplo anterior, a instrução `using B::i` faz com que um segundo `int i` seja declarado na função `g()`. A instrução `using B::f` não entra em conflito com a função `f(char)` porque os nomes de função introduzidos por `B::f` têm tipos de parâmetro diferentes.

Exemplo

Uma declaração de função local não pode ter o mesmo nome e tipo que uma função introduzida usando a declaração. Por exemplo:

```
// functions_in_namespaces2.cpp
// C2668 expected
namespace B {
    void f(int);
    void f(double);
}

namespace C {
    void f(int);
    void f(double);
    void f(char);
}

void h() {
    using B::f;        // introduces B::f(int) and B::f(double)
    using C::f;        // C::f(int), C::f(double), and C::f(char)
    f('h');           // calls C::f(char)
    f(1);             // C2668 ambiguous: B::f(int) or C::f(int)?
    void f(int);       // C2883 conflicts with B::f(int) and C::f(int)
}
```

Exemplo

No que diz respeito à herança, quando uma declaração using introduz um nome de uma classe base em um escopo de classe derivada, as funções de membro na classe derivada substituem funções de membro virtuais com o mesmo nome e tipo de argumento na classe base.

```

// using_declaration_inheritance1.cpp
#include <stdio.h>
struct B {
    virtual void f(int) {
        printf_s("In B::f(int)\n");
    }

    virtual void f(char) {
        printf_s("In B::f(char)\n");
    }

    void g(int) {
        printf_s("In B::g\n");
    }

    void h(int);
};

struct D : B {
    using B::f;
    void f(int) { // ok: D::f(int) overrides B::f(int)
        printf_s("In D::f(int)\n");
    }

    using B::g;
    void g(char) { // ok: there is no B::g(char)
        printf_s("In D::g(char)\n");
    }

    using B::h;
    void h(int) {} // Note: D::h(int) hides non-virtual B::h(int)
};

void f(D* pd) {
    pd->f(1); // calls D::f(int)
    pd->f('a'); // calls B::f(char)
    pd->g(1); // calls B::g(int)
    pd->g('a'); // calls D::g(char)
}

int main() {
    D * myd = new D();
    f(myd);
}

```

```

In D::f(int)
In B::f(char)
In B::g
In D::g(char)

```

Exemplo

Todas as instâncias de um nome mencionado em uma declaração using devem estar acessíveis. Em particular, se uma classe derivada usar uma declaração using para acessar um membro de uma classe base, o nome do membro deverá ser acessível. Se o nome for o de uma função de membro sobreescrita, todas as funções nomeadas deverão estar acessíveis.

Para obter mais informações sobre acessibilidade de membros, consulte [controle de acesso de membro](#).

```
// using_declaration_inheritance2.cpp
// C2876 expected
class A {
private:
    void f(char);
public:
    void f(int);
protected:
    void g();
};

class B : public A {
    using A::f;    // C2876: A::f(char) is inaccessible
public:
    using A::g;    // B::g is a public synonym for A::g
};
```

Confira também

[Namespaces](#)

[Palavras-chave](#)

volatile (C++)

02/09/2020 • 6 minutes to read • [Edit Online](#)

Um qualificador de tipo que você pode usar para declarar que um objeto pode ser modificado no programa pelo hardware.

Sintaxe

```
volatile declarator ;
```

Comentários

Você pode usar a opção de compilador [/volatile](#) para modificar como o compilador interpreta essa palavra-chave.

O Visual Studio interpreta a `volatile` palavra-chave de forma diferente, dependendo da arquitetura de destino. Para o ARM, se nenhuma opção de compilador `/volatile` for especificada, o compilador executará como se `/volatile: ISO` fosse especificado. Para arquiteturas diferentes de ARM, se nenhuma opção de compilador `/volatile` for especificada, o compilador executará como se `/volatile: MS` fosse especificado; Portanto, para arquiteturas diferentes do ARM, é altamente recomendável que você especifique `/volatile: ISO` e use primitivos de sincronização explícitos e intrínsecos do compilador quando estiver lidando com a memória que é compartilhada entre threads.

Você pode usar o `volatile` qualificador para fornecer acesso a locais de memória que são usados por processos assíncronos, como manipuladores de interrupção.

Quando `volatile` é usado em uma variável que também tem a palavra-chave `__restrict`, `volatile` tem precedência.

Se um `struct` membro for marcado como `volatile`, `volatile` será propagado para toda a estrutura. Se uma estrutura não tiver um comprimento que possa ser copiado na arquitetura atual usando uma instrução, `volatile` pode ser completamente perdido nessa estrutura.

A `volatile` palavra-chave pode não ter nenhum efeito em um campo se uma das seguintes condições for verdadeira:

- O tamanho do campo volátil excede o tamanho máximo que pode ser copiado na arquitetura atual usando uma instrução.
- O comprimento do conteúdo mais externo que contém `struct` (ou se for um membro de um possivelmente aninhado `struct`) excede o tamanho máximo que pode ser copiado na arquitetura atual usando uma instrução.

Embora o processador não reordene os acessos de memória não armazenáveis em cache, as variáveis não armazenáveis em cache devem ser marcadas como `volatile` para garantir que o compilador não reordene os acessos de memória.

Os objetos declarados como `volatile` não são usados em determinadas otimizações porque seus valores podem ser alterados a qualquer momento. O sistema sempre lê o valor atual de um objeto volátil quando solicitado, mesmo que uma instrução anterior tenha solicitado um valor do mesmo objeto. Além disso, o valor do objeto é gravado imediatamente na atribuição.

Compatível com ISO

Se você estiver familiarizado com a palavra-chave volátil do C# ou estiver familiarizado com o comportamento do `volatile` em versões anteriores do compilador do Microsoft C++ (MSVC), lembre-se de que a palavra-chave ISO 11 do C++ `volatile` é diferente e tem suporte em MSVC quando a opção de compilador `/volatile:ISO` for especificada. (Para ARM, ela é especificada por padrão). A `volatile` palavra-chave no código padrão ISO 11 do C++ é ser usada somente para acesso de hardware; não a use para comunicação entre threads. Para comunicação entre threads, use mecanismos como `std::Atomic <T>` da [biblioteca padrão C++](#).

Fim de compatível com ISO

Específico da Microsoft

Quando a opção de compilador `/volatile:MS` é usada – por padrão, quando há arquiteturas diferentes de ARM, o compilador gera código extra para manter a ordenação entre referências a objetos voláteis, além de manter a ordenação para referências a outros objetos globais. Especialmente:

- Uma gravação em um objeto volátil (também conhecida como gravação volátil) tem semântica de versão; ou seja, uma referência a um objeto global ou estático que ocorre antes de uma gravação em um objeto volátil na sequência da instrução ocorrerá antes dessa gravação volátil no binário compilado.
- Uma leitura de um objeto volátil (também conhecida como leitura volátil) tem semântica de aquisição; ou seja, uma referência a um objeto global ou estático que ocorre depois de uma leitura de memória volátil na sequência da instrução ocorrerá depois dessa leitura volátil no binário compilado.

Isso permite que os objetos voláteis sejam usados para versões e bloqueios de memória em aplicativos multithread.

NOTE

Quando ele se baseia na garantia avançada que é fornecida quando a opção de compilador `/volatile:MS` é usada, o código não é portátil.

FINAL específico da Microsoft

Confira também

[Palavras-chave](#)

`const`

[Ponteiros const e voláteis](#)

decltype (C++)

02/09/2020 • 10 minutes to read • [Edit Online](#)

O `decltype` especificador de tipo produz o tipo de uma expressão especificada. O `decltype` especificador de tipo, junto com a `auto` palavra-chave, é útil principalmente para os desenvolvedores que gravam bibliotecas de modelos. Use `auto` e `decltype` para declarar uma função de modelo cujo tipo de retorno depende dos tipos de seus argumentos de modelo. Ou use `auto` e `decltype` para declarar uma função de modelo que encapsula uma chamada para outra função e, em seguida, retorna o tipo de retorno da função encapsulada.

Sintaxe

```
decltype( ***expressão* de** )**
```

Parâmetros

expressão

Uma expressão. Para obter mais informações, consulte [expressões](#).

Valor retornado

O tipo do parâmetro da *expressão*.

Comentários

O `decltype` especificador de tipo tem suporte no Visual Studio 2010 ou em versões posteriores e pode ser usado com código nativo ou gerenciado. `decltype(auto)` (C++ 14) tem suporte no Visual Studio 2015 e posterior.

O compilador usa as regras a seguir para determinar o tipo do parâmetro de *expressão*.

- Se o parâmetro *expression* for um identificador ou um [acesso de membro de classe](#), `decltype(expression)` será o tipo da entidade chamada por *expressão*. Se não houver nenhuma entidade ou o parâmetro de *expressão* nomeia um conjunto de funções sobrecarregadas, o compilador produzirá uma mensagem de erro.
- Se o parâmetro *expression* for uma chamada para uma função ou uma função de operador sobrecarregada, `decltype(expression)` será o tipo de retorno da função. Os parênteses em torno de um operador sobrecarregado são ignorados.
- Se o parâmetro *expression* for um [rvalue](#), `decltype(expression)` será o tipo de *expressão*. Se o parâmetro *expression* for um [lvalue](#), `decltype(expression)` será uma [referência lvalue](#) para o tipo de *expressão*.

O exemplo de código a seguir demonstra alguns usos do `decltype` especificador de tipo. Primeiro, suponha que você tenha codificado as instruções a seguir.

```
int var;
const int&& fx();
struct A { double x; }
const A* a = new A();
```

Em seguida, examine os tipos que são retornados pelas quatro `decltype` instruções na tabela a seguir.

DE	TYPE	OBSERVAÇÕES
<code>decltype(fx());</code>	<code>const int&&</code>	Uma referência rvalue a um <code>const int</code> .
<code>decltype(var);</code>	<code>int</code>	O tipo da variável <code>var</code> .
<code>decltype(a->x);</code>	<code>double</code>	O tipo do acesso de membro.
<code>decltype((a->x));</code>	<code>const double&</code>	Os parênteses internos fazem com que a instrução seja avaliada como uma expressão em vez de um acesso de membro. E como <code>a</code> é declarado como um <code>const</code> ponteiro, o tipo é uma referência a <code>const double</code> .

Decltype e auto

No C++ 14, você pode usar sem `decltype(auto)` nenhum tipo de retorno à direita para declarar uma função de modelo cujo tipo de retorno depende dos tipos de seus argumentos de modelo.

No C++ 11, você pode usar o `decltype` especificador de tipo em um tipo de retorno à direita, junto com a `auto` palavra-chave, para declarar uma função de modelo cujo tipo de retorno depende dos tipos de seus argumentos de modelo. Por exemplo, considere o exemplo de código a seguir, no qual o tipo de retorno da função de modelo depende dos tipos dos argumentos de modelo. No exemplo de código, o espaço reservado *desconhecido* indica que o tipo de retorno não pode ser especificado.

```
template<typename T, typename U>
UNKNOWN func(T&& t, U&& u){ return t + u; };
```

A introdução do `decltype` especificador de tipo permite que um desenvolvedor obtenha o tipo da expressão que a função de modelo retorna. Use a *sintaxe de declaração de função alternativa* que é mostrada posteriormente, a `auto` palavra-chave e o `decltype` especificador de tipo para declarar um tipo de retorno especificado de forma *tardia*. O tipo de retorno com especificação tardia é determinado quando a declaração é compilada, não quando ela é codificada.

O protótipo a seguir ilustra a sintaxe de uma declaração de função alternativa. Observe que os `const` `volatile` qualificadores e a `throw` especificação de exceção são opcionais. O espaço reservado *function_body* representa uma instrução composta que especifica o que a função faz. Como uma melhor prática de codificação, o espaço reservado da *expressão* na `decltype` instrução deve corresponder à expressão especificada pela `return` instrução, se houver, no *function_body*.

```
auto ***function_name* ( parametersopt ) constopt volatileopt -> decltype( expression )  
noexceptopt { function_body de expressão de consentimento de aceitação de parâmetros** } ; **
```

No exemplo de código a seguir, o tipo de retorno com especificação tardia da função de modelo `myFunc` é determinado pelos tipos dos argumentos de modelo `t` e `u`. Como uma melhor prática de codificação, o exemplo de código também usa referências a rvalue e o `forward` modelo de função, que oferece suporte ao *encaminhamento perfeito*. Para obter mais informações, consulte [Declarador de referência Rvalue: &&](#).

```
//C++11
template<typename T, typename U>
auto myFunc(T&& t, U&& u) -> decltype (forward<T>(t) + forward<U>(u))
    { return forward<T>(t) + forward<U>(u); }

//C++14
template<typename T, typename U>
decltype(auto) myFunc(T&& t, U&& u)
    { return forward<T>(t) + forward<U>(u); }
```

Funções decltype e Forwarding (C++ 11)

As funções de encaminhamento encapsulam chamadas para outras funções. Considere um modelo de função que encaminha seus argumentos, ou os resultados de uma expressão que envolve esses argumentos, para outra função. Além disso, a função de encaminhamento retorna o resultado da chamada para a outra função. Nesse cenário, o tipo de retorno da função de encaminhamento deve ser igual ao tipo de retorno da função encapsulada.

Nesse cenário, você não pode gravar uma expressão de tipo apropriada sem o `decltype` especificador de tipo. O `decltype` especificador de tipo habilita funções de encaminhamento genéricas porque não perde as informações necessárias sobre se uma função retorna um tipo de referência. Para obter um exemplo de código de uma função de encaminhamento, consulte o exemplo anterior da função de modelo `myFunc`.

Exemplo

O exemplo de código a seguir declara o tipo de retorno com especificação tardia da função de modelo `Plus()`. A `Plus` função processa seus dois operandos com a `operator+` sobrecarga. Consequentemente, a interpretação do operador de adição (`+`) e o tipo de retorno da `Plus` função dependem dos tipos dos argumentos da função.

```

// decltype_1.cpp
// compile with: cl /EHsc decltype_1.cpp

#include <iostream>
#include <string>
#include <utility>
#include <iomanip>

using namespace std;

template<typename T1, typename T2>
auto Plus(T1&& t1, T2&& t2) ->
    decltype(forward<T1>(t1) + forward<T2>(t2))
{
    return forward<T1>(t1) + forward<T2>(t2);
}

class X
{
    friend X operator+(const X& x1, const X& x2)
    {
        return X(x1.m_data + x2.m_data);
    }

public:
    X(int data) : m_data(data) {}
    int Dump() const { return m_data; }
private:
    int m_data;
};

int main()
{
    // Integer
    int i = 4;
    cout <<
        "Plus(i, 9) = " <<
        Plus(i, 9) << endl;

    // Floating point
    float dx = 4.0;
    float dy = 9.5;
    cout <<
        setprecision(3) <<
        "Plus(dx, dy) = " <<
        Plus(dx, dy) << endl;

    // String
    string hello = "Hello, ";
    string world = "world!";
    cout << Plus(hello, world) << endl;

    // Custom type
    X x1(20);
    X x2(22);
    X x3 = Plus(x1, x2);
    cout <<
        "x3.Dump() = " <<
        x3.Dump() << endl;
}

```

```

Plus(i, 9) = 13
Plus(dx, dy) = 13.5
Hello, world!
x3.Dump() = 42

```

Exemplo

Visual Studio 2017 e posterior: O compilador analisa `decltype` argumentos quando os modelos são declarados em vez de instanciados. Consequentemente, se uma especialização não dependente for encontrada no `decltype` argumento, ela não será adiada para tempo de instanciação e será processada imediatamente e quaisquer erros resultantes serão diagnosticados nesse momento.

O exemplo a seguir mostra esse erro do compilador gerado no momento da declaração:

```
#include <utility>
template <class T, class ReturnT, class... ArgsT> class IsCallable
{
public:
    struct BadType {};
    template <class U>
    static decltype(std::declval<T>()(std::declval<ArgsT>(...))) Test(int); //C2064. Should be declval<U>
    template <class U>
    static BadType Test(...);
    static constexpr bool value = std::is_convertible<decltype(Test<T>()), ReturnT>::value;
};

constexpr bool test1 = IsCallable<int(), int>::value;
static_assert(test1, "PASS1");
constexpr bool test2 = !IsCallable<int*, int>::value;
static_assert(test2, "PASS2");
```

Requisitos

Visual Studio 2010 ou versões posteriores.

`decltype(auto)` requer o Visual Studio 2015 ou posterior.

Atributos em C++

02/09/2020 • 8 minutes to read • [Edit Online](#)

O padrão C++ define um conjunto de atributos e também permite que os fornecedores de compilador definam seus próprios atributos (dentro de um namespace específico do fornecedor), mas os compiladores são necessários para reconhecer apenas os atributos definidos no padrão.

Em alguns casos, os atributos padrão se sobrepõem com parâmetros `declspec` específicos do compilador. No Visual C++, você pode usar o `[[deprecated]]` atributo em vez de usar `declspec(deprecated)` e o atributo será reconhecido por qualquer compilador em conformidade. Para todos os outros parâmetros de `declspec`, como `DllImport` e `dllexport`, ainda não há nenhum equivalente de atributo, portanto, você deve continuar a usar a sintaxe `declspec`. Os atributos não afetam o sistema de tipos e não alteram o significado de um programa. Os compiladores ignoram valores de atributo que não reconhecem.

Visual Studio 2017 versão 15,3 e posterior (disponível com `/std: c++ 17`): no escopo de uma lista de atributos, você pode especificar o namespace para todos os nomes com um único `using` apresentador:

```
void g() {
    [[using rpr: kernel, target(cpu,gpu)]] // equivalent to [[ rpr::kernel, rpr::target(cpu,gpu) ]]
    do task();
}
```

Atributos padrão do C++

No C++ 11, os atributos fornecem uma maneira padronizada para anotar construções do C++ (incluindo, mas não se limitando a classes, funções, variáveis e blocos) com informações adicionais que podem ou não ser específicas do fornecedor. Um compilador pode usar essas informações para gerar mensagens informativas ou para aplicar uma lógica especial ao compilar o código atribuído. O compilador ignora todos os atributos que ele não reconhece, o que significa que você não pode definir seus próprios atributos personalizados usando essa sintaxe. Os atributos são colocados entre colchetes duplos:

```
[[deprecated]]
void Foo(int);
```

Os atributos representam uma alternativa padronizada para extensões específicas de fornecedor, como `#pragma` diretivas, `__declspec ()` (Visual C++) ou GNU (`__attribute__`). No entanto, ainda será necessário usar as construções específicas do fornecedor para a maioria das finalidades. O padrão atualmente especifica os seguintes atributos que um compilador de conformidade deve reconhecer:

- `[[noreturn]]` Especifica que uma função nunca retorna; em outras palavras, ele sempre gera uma exceção. O compilador pode ajustar suas regras de compilação para `[[noreturn]]` entidades.
- `[[carries_dependency]]` Especifica que a função propaga a ordem de dependência de dados em relação à sincronização de threads. O atributo pode ser aplicado a um ou mais parâmetros, para especificar que o argumento passado transporta uma dependência no corpo da função. O atributo pode ser aplicado à própria função, para especificar que o valor de retorno transporta uma dependência da função. O compilador pode usar essas informações para gerar um código mais eficiente.
- `[[deprecated]]` **Visual Studio 2015 e posterior:** Especifica que uma função não deve ser usada e pode não existir em versões futuras de uma interface de biblioteca. O compilador pode usar isso para gerar uma

mensagem informativa quando o código do cliente tenta chamar a função. Pode ser aplicado à declaração de uma classe, um nome de typedef, uma variável, um membro de dados não estático, uma função, um namespace, uma enumeração, um enumerador ou uma especialização de modelo.

- **[[fallthrough]] Visual Studio 2017 e posterior:** (disponível com [/std: c++ 17](#)) o **[[fallthrough]]** atributo pode ser usado no contexto de instruções `switch` como uma dica para o compilador (ou qualquer pessoa que lê o código) que o comportamento de fallthrough é pretendido. No momento, o compilador do Microsoft C++ não avisa sobre o comportamento do fallthrough, portanto, esse atributo não tem efeito sobre o comportamento do compilador.
- **[[nodiscard]] Visual Studio 2017 versão 15,3 e posterior:** (disponível com [/std: c++ 17](#)) especifica que o valor de retorno de uma função não deve ser Descartado. Gera C4834 de aviso, conforme mostrado neste exemplo:

```
[[nodiscard]]  
int foo(int i) { return i * i; }  
  
int main()  
{  
    foo(42); //warning C4834: discarding return value of function with 'nodiscard' attribute  
    return 0;  
}
```

- **[[maybe_unused]] Visual Studio 2017 versão 15,3 e posteriores:** (disponível com [/std: c++ 17](#)) especifica que uma especialização de variável, função, classe, typedef, membro de dados não estático, enumeração ou modelo pode não ser usada intencionalmente. O compilador não avisa quando uma entidade marcada **[[maybe_unused]]** não é usada. Uma entidade que é declarada sem o atributo pode ser redeclarada posteriormente com o atributo e vice-versa. Uma entidade é considerada marcada depois que sua primeira declaração marcada é analisada e para o restante da tradução da unidade de tradução atual.

Atributos específicos da Microsoft

- **[[gsl::suppress(rules)]]** Este atributo específico da Microsoft é usado para suprimir avisos de verificadores que impõem regras de [GSL \(biblioteca de suporte de diretrizes\)](#) no código. Por exemplo, considere este trecho de código:

```
int main()  
{  
    int arr[10]; // GSL warning C26494 will be fired  
    int* p = arr; // GSL warning C26485 will be fired  
    [[gsl::suppress(bounds.1)]] // This attribute suppresses Bounds rule #1  
    {  
        int* q = p + 1; // GSL warning C26481 suppressed  
        p = q--; // GSL warning C26481 suppressed  
    }  
}
```

O exemplo gera estes avisos:

- 26494 (tipo de regra 5: sempre inicializar um objeto.)
- 26485 (Associação regra 3: nenhuma matriz para ponteiro decaimento.)
- 26481 (regra de limites 1: não use aritmética de ponteiro. Use span em vez disso.)

Os dois primeiros avisos são acionados quando você compila esse código com a ferramenta de análise de código do CppCoreCheck instalada e ativada. Mas o terceiro aviso não é acionado devido ao atributo. Você pode suprimir o perfil de limites inteiro escrevendo **[[GSL:: suprimir (limites)]]** sem incluir um número de

regra específico. As Diretrizes Principais do C++ são projetadas para ajudá-lo a escrever um código melhor e mais seguro. O atributo suprimir facilita a desativação dos avisos quando eles não são desejados.

Operadores, precedência e Associação internos do C++

02/09/2020 • 5 minutes to read • [Edit Online](#)

A linguagem C++ inclui todos os operadores C e adiciona vários operadores novos. Os operadores especificam uma avaliação a ser executada em um ou mais operandos.

Precedência e Associação

Precedência de operador especifica a ordem das operações em expressões que contêm mais de um operador. A *Associação* de operador especifica se, em uma expressão que contém vários operadores com a mesma precedência, um operando é agrupado com aquele à esquerda ou aquele à direita.

Ortografias alternativas

O C++ especifica grafias alternativas para alguns operadores. Em C, as grafias alternativas são fornecidas como macros no <iso646.h> cabeçalho. Em C++, essas alternativas são palavras-chave e o uso do <iso646.h> ou do equivalente em C++ <ciso646> é preterido. No Microsoft C++, a `/permissive-` opção ou do `/Za` compilador é necessária para habilitar as grafias alternativas.

Tabela de precedência de operador C++ e Associação

A tabela a seguir mostra a precedência e a associatividade dos operadores C++ (da precedência mais alta a mais baixa). Os operadores com o mesmo número de precedência têm igual precedência, a menos que outra relação seja explicitamente forçada por parênteses.

Descrição do operador	Operador	Alternativa
Precedência de grupo 1, sem Associação		
Resolução de escopo	<code>::</code>	
Prioridade do grupo 2, associação da esquerda para a direita		
Seleção de membro (objeto ou ponteiro)	<code>.</code> or <code>-></code>	
Subscrito de matriz	<code>[]</code>	
Chamada de função	<code>()</code>	
Incremento de sufixo	<code>++</code>	
Decremento de sufixo	<code>--</code>	

DESCRÍÇÃO DO OPERADOR	OPERADOR	ALTERNATIVA
Nome do tipo	<code>typeid</code>	
Conversão de tipo constante	<code>const_cast</code>	
Conversão de tipo dinâmico	<code>dynamic_cast</code>	
Conversão de tipo reinterpretado	<code>reinterpret_cast</code>	
Conversão de tipo estático	<code>static_cast</code>	
Prioridade do grupo 3, associação direita à esquerda		
Tamanho do objeto ou tipo	<code>sizeof</code>	
Incremento de prefixo	<code>++</code>	
Decremento de prefixo	<code>--</code>	
Complemento de um	<code>~</code>	<code>compl</code>
Não lógico	<code>!</code>	<code>not</code>
Negação unária	<code>-</code>	
Adição de unário	<code>+</code>	
Address-of	<code>&</code>	
Indireção	<code>*</code>	
Criar objeto	<code>new</code>	
Objeto Destroy	<code>delete</code>	
Vertida	<code>()</code>	
Prioridade do grupo 4, associação da esquerda para a direita		
Ponteiro para membro (objetos ou ponteiros)	<code>.*</code> or <code>->*</code>	
Prioridade do grupo 5, associação da esquerda para a direita		
Multiplicação	<code>*</code>	

DESCRÍÇÃO DO OPERADOR	OPERADOR	ALTERNATIVA
Divisão	/	
Modulus	%	
Prioridade do grupo 6, associação da esquerda para a direita		
Adição	+	
Subtração	-	
Prioridade do grupo 7, associação da esquerda para a direita		
Deslocamento à esquerda	<<	
Deslocamento à direita	>>	
Prioridade do grupo 8, associação da esquerda para a direita		
Menor que	<	
Maior que	>	
Menor ou igual a	<=	
Maior ou igual a	>=	
Prioridade do grupo 9, associação da esquerda para a direita		
Igualitário	==	
Desigualdade	!=	not_eq
Prioridade do grupo 10 esquerda para direita		
AND bit a bit	&	bitand
Prioridade do grupo 11, associação da esquerda para a direita		
OR exclusivo bit a bit	^	xor

DESCRÍÇÃO DO OPERADOR	OPERADOR	ALTERNATIVA
Prioridade do grupo 12, da esquerda para a direita		
OR inclusivo bit a bit		bitor
Prioridade do grupo 13, da esquerda para a direita		
AND lógico	&&	and
Prioridade do grupo 14, associação da esquerda para a direita		
OR lógico		or
Prioridade do grupo 15, associação direita à esquerda		
Condicional	? :	
Associação de grupo 16, prioridade direita para esquerda		
Atribuição	=	
Atribuição de multiplicação	*=	
Atribuição de divisão	/=	
Atribuição de módulo	%=	
Atribuição de adição	+=	
Atribuição de subtração	-=	
Atribuição de shift esquerda	<<=	
Atribuição de shift direita	>>=	
Atribuição AND de bit a bit	&=	and_eq
Atribuição OR inclusivo de bit a bit	=	or_eq
Atribuição OR exclusivo de bit a bit	^=	xor_eq
Prioridade de grupo 17, associação direita para esquerda		
expressão de lançamento	throw	

DESCRIÇÃO DO OPERADOR	OPERADOR	ALTERNATIVA
Prioridade de grupo 18, associação da esquerda para a direita		
Pontos	,	

Confira também

[Sobrecarga de operador](#)

Operador alignof

02/09/2020 • 2 minutes to read • [Edit Online](#)

O `alignof` operador retorna o alinhamento em bytes do tipo especificado como um valor do tipo `size_t`.

Sintaxe

```
alignof( type )
```

Comentários

Por exemplo:

EXPRESSION	VALOR
<code>alignof(char)</code>	1
<code>alignof(short)</code>	2
<code>alignof(int)</code>	4
<code>alignof(long long)</code>	8
<code>alignof(float)</code>	4
<code>alignof(double)</code>	8

O `alignof` valor é o mesmo que o valor para `sizeof` para tipos básicos. Considere, no entanto, este exemplo:

```
typedef struct { int a; double b; } S;
// alignof(S) == 8
```

Nesse caso, o `alignof` valor é o requisito de alinhamento do maior elemento na estrutura.

De maneira semelhante, para

```
typedef __declspec(align(32)) struct { int a; } S;
```

`alignof(S)` é igual a `32`.

Um uso para `alignof` seria como um parâmetro para uma de suas próprias rotinas de alocação de memória. Por exemplo, dada a seguinte estrutura definida `S`, você poderia chamar uma rotina de alocação de memória de nome `aligned_malloc` para alocar memória em um limite de alinhamento específico.

```
typedef __declspec(align(32)) struct { int a; double b; } S;
int n = 50; // array size
S* p = (S*)aligned_malloc(n * sizeof(S), alignof(S));
```

Para obter mais informações sobre como modificar o alinhamento, consulte:

- [pack](#)
- [alinha](#)
- [__unaligned](#)
- [/ZP \(alinhamento de membro de struct\)](#)
- [Exemplos de alinhamento de estrutura](#) (específico para x64)

Para obter mais informações sobre as diferenças no alinhamento no código para x86 e x64, consulte:

- [conflitos com o compilador x86](#)

Específico da Microsoft

`alignof` e `__alignof` são sinônimos no compilador da Microsoft. Antes de se tornar parte do padrão no C++ 11, o operador específico da Microsoft `__alignof` fornecia essa funcionalidade. Para portabilidade máxima, você deve usar o `alignof` operador em vez do operador específico da Microsoft `__alignof`.

Para compatibilidade com versões anteriores, `__alignof` é um sinônimo para `alignof` a menos que a opção do compilador [/za](#) (desabilite extensões de linguagem) seja especificada.

Confira também

[Expressões com operadores unários](#)

[Palavras-chave](#)

Operador `__uuidof`

02/09/2020 • 2 minutes to read • [Edit Online](#)

Específico da Microsoft

Recupera o GUID vinculado à expressão.

Sintaxe

```
__uuidof ( ***expressão* de** ) **
```

Comentários

A *expressão* pode ser um nome de tipo, um ponteiro, uma referência ou uma matriz desse tipo, um modelo especializado nesses tipos ou uma variável desses tipos. O argumento é válido enquanto o compilador puder usá-lo para localizar o GUID vinculado.

Um caso especial desse intrínseco é quando 0 ou NULL é fornecido como o argumento. Nesse caso, o `__uuidof` retornará um GUID formado por zeros.

Use essa palavra-chave para extrair o GUID vinculado a:

- Um objeto pelo `uuid` atributo estendido.
- Um bloco de biblioteca criado com o `module` atributo.

NOTE

Em uma compilação de depuração, o `__uuidof` sempre inicializa um objeto dinamicamente (em tempo de execução). Em uma compilação de versão, o `__uuidof` pode, estaticamente (no momento da compilação) inicializar um objeto.

Para compatibilidade com versões anteriores, `__uuidof` é um sinônimo para `__uuidof` a menos que a opção do compilador `/Za` (desabilite extensões de linguagem) seja especificada.

Exemplo

O código a seguir (compilado com ole32.lib) exibirá o uuid de um bloco de biblioteca criado com o atributo `module`:

```
// expre_uuidof.cpp
// compile with: ole32.lib
#include <stdio.h>
#include <windows.h>

[emitidl];
[module(name="MyLib")];
[export]
struct stuff {
    int i;
};

int main() {
    LPOLESTR lpolestr;
    StringFromCLSID(__uuidof(MyLib), &lpolestr);
    wprintf_s(L"%s", lpolestr);
    CoTaskMemFree(lpolestr);
}
```

Comentários

Nos casos em que o nome da biblioteca não está mais no escopo, você pode usar `__LIBID_` em vez de `__uuidof` . Por exemplo:

```
StringFromCLSID(__LIBID_, &lpolestr);
```

FINAL específico da Microsoft

Confira também

[Expressões com operadores unários](#)

[Palavras-chave](#)

Operadores aditivos: + e -

15/04/2020 • 5 minutes to read • [Edit Online](#)

Sintaxe

```
expression + expression
expression - expression
```

Comentários

Os operadores de adição são:

- Adição**+()
- Subtração-()

Esses operadores binários possuem associatividade da esquerda para a direita.

Os operadores de adição usam operandos de tipos aritméticos ou ponteiro. O resultado do**+** operador de adição é a soma dos operandos. O resultado do operador- de subtração é a diferença entre os operandos. Se um ou ambos os operandos forem ponteiros, eles devem ser ponteiros para os objetos, não para funções. Se ambos os operandos forem ponteiros, os resultados não serão significativos a menos que ambos sejam ponteiros para os objetos na mesma matriz.

Os operadores aditivos tomam operandos de *tipos aritméticos, integrase escalares*. Eles são definidos na tabela a seguir.

Tipos usados com operadores de adição

TYPE	SIGNIFICADO
<i>Aritmética</i>	Os tipos integral e flutuante são coletivamente chamados de tipos "aritméticos".
<i>integral</i>	Os tipos char e int de todos os tamanhos (curto, longo) e enumerações são os tipos "integrais".
<i>escalar</i>	Os operandos escalares são operandos aritméticos ou de ponteiro.

As combinações legais para usar esses operadores são:

*arithmetic**aritmética aritmética +*

escalado + integral

escalain integral + scalar

*arithmetic**aritmética aritmética -*

escalar - escalar

Observe que a adição e subtração não são operações equivalentes.

Se ambos os operandos forem do tipo aritmética, as conversões cobertas pelas [Conversões Padrão](#) são aplicadas aos operandos, e o resultado é do tipo convertido.

Exemplo

```
// expre_Additive_Operators.cpp
// compile with: /EHsc
#include <iostream>
#define SIZE 5
using namespace std;
int main() {
    int i = 5, j = 10;
    int n[SIZE] = { 0, 1, 2, 3, 4 };
    cout << "5 + 10 = " << i + j << endl
        << "5 - 10 = " << i - j << endl;

    // use pointer arithmetic on array

    cout << "n[3] = " << *( n + 3 ) << endl;
}
```

Adição de ponteiro

Se um dos operandos em uma operação de adição é um ponteiro para uma matriz de objetos, o outro deve ser do tipo integral. O resultado é um ponteiro que tem o mesmo tipo do ponteiro original e que aponta para outro elemento da matriz. O fragmento de código a seguir ilustra esse conceito:

```
short IntArray[10]; // Objects of type short occupy 2 bytes
short *pIntArray = IntArray;

for( int i = 0; i < 10; ++i )
{
    *pIntArray = i;
    cout << *pIntArray << "\n";
    pIntArray = pIntArray + 1;
}
```

Embora o valor de integral 1 seja adicionado a `pIntArray`, isso não significa "adicionar 1 ao endereço"; significa "ajustar o ponteiro de modo a apontar para o próximo objeto na matriz" que esteja afastado por 2 bytes (ou `sizeof(int)`).

NOTE

O código do formato `pIntArray = pIntArray + 1` raramente é encontrado em programas C++; para executar um incremento, estes formatos são preferíveis: `pIntArray++` ou `pIntArray += 1`.

Subtração de ponteiro

Se ambos os operandos forem ponteiros, o resultado da subtração será a diferença (em elementos de matriz) entre os operandos. A expressão de subtração produz `ptrdiff_t` um resultado integral assinado <do tipo definido no padrão incluir arquivo stddef.h>.

Um dos operandos pode ser do tipo integral, desde que seja o segundo operando. O resultado da subtração é do mesmo tipo do ponteiro original. O valor da subtração é um ponteiro para o elemento de matriz ($n - i$)th, onde n é o elemento apontado pelo ponteiro original e i é o valor integral do segundo operando.

Confira também

[Expressões com operadores binários](#)

[Operadores internos C++, precedência e associatividade](#)

[Operadores aditivos C](#)

Operador de endereço:&

02/09/2020 • 4 minutes to read • [Edit Online](#)

Sintaxe

```
& cast-expression
```

Comentários

O operador address-of unário (`&`) Obtém o endereço de seu operando. O operando do operador address-of pode ser um designador de função ou um l-Value que designa um objeto que não seja um campo de bits.

O operador address-of pode ser aplicado somente a variáveis com tipos fundamentais, de estrutura, classe ou união que são declarados no nível de escopo do arquivo, ou a referências de matriz subscrita. Nessas expressões, uma expressão constante que não inclui o operador address-of pode ser adicionada ou subtraída da expressão address-of.

Quando aplicado a funções ou valores l, o resultado da expressão será um tipo de ponteiro (um valor r) derivado do tipo do operando. Por exemplo, se o operando for do tipo `char`, o resultado da expressão será do tipo ponteiro para `char`. O operador address-of, aplicado a `const` ou `volatile` Objects, é avaliado `const type *` como `volatile type *` ou, onde `Type` é o tipo do objeto original.

Quando o operador address-of é aplicado a um nome qualificado, o resultado depende se o *nome qualificado* especifica um membro estático. Nesse caso, o resultado é um ponteiro para o tipo especificado na declaração do membro. Se o membro não for estático, o resultado será um ponteiro para o *nome* do membro da classe indicada por *nome de classe qualificado*. (Consulte as [expressões primárias](#) para obter mais informações sobre *nome de classe qualificado*.) O fragmento de código a seguir mostra como o resultado difere, dependendo se o membro é estático:

```
// expe_Address_Of_Operator.cpp
// C2440 expected
class PTM {
public:
    int iValue;
    static float fValue;
};

int main() {
    int PTM::*piValue = &PTM::iValue; // OK: non-static
    float PTM::*pfValue = &PTM::fValue; // C2440 error: static
    float *spfValue     = &PTM::fValue; // OK
}
```

Neste exemplo, a expressão `&PTM::fValue` gera o tipo `float *` em vez do tipo `float PTM::*` porque `fValue` é um membro estático.

O endereço de uma função sobreporregada pode ser obtido somente quando está claro que versão da função está sendo referenciada. Consulte [sobreporregada de função](#) para obter informações sobre como obter o endereço de uma função sobreporregada específica.

Aplicar o operador address-of a um tipo de referência fornece o mesmo resultado que aplicar o operador ao objeto ao qual a referência está associada. Por exemplo:

Exemplo

```
// expre_Address_Of_Operator2.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;
int main() {
    double d;           // Define an object of type double.
    double& rd = d;   // Define a reference to the object.

    // Obtain and compare their addresses
    if( &d == &rd )
        cout << "&d equals &rd" << endl;
}
```

Saída

```
&d equals &rd
```

O exemplo a seguir usa o operador address-of para passar um argumento de ponteiro para uma função:

```
// expre_Address_Of_Operator3.cpp
// compile with: /EHsc
// Demonstrate address-of operator &

#include <iostream>
using namespace std;

// Function argument is pointer to type int
int square( int *n ) {
    return (*n) * (*n);
}

int main() {
    int mynum = 5;
    cout << square( &mynum ) << endl;    // pass address of int
}
```

Saída

```
25
```

Confira também

[Expressões com operadores unários](#)

[Operadores internos C++, precedência e associatividade](#)

[Declarador de referência Lvalue: &](#)

[Operadores de endereço e de indireção](#)

Operadores de atribuição

02/09/2020 • 10 minutes to read • [Edit Online](#)

Sintaxe

expression atribuição de expressão- expressão do operador

assignment-operator: one of

`=` `*=` `/=` `%=` `+=` `-=` `<=` `>=` `&=` `^=` `|=`

Comentários

Operadores de atribuição armazenam um valor no objeto especificado pelo operando esquerdo. Há dois tipos de operações de atribuição:

- *atribuição simples*, na qual o valor do segundo operando é armazenado no objeto especificado pelo primeiro operando.
- *atribuição composta*, na qual uma operação aritmética, de deslocamento ou bit-de-bits é executada antes de armazenar o resultado.

Todos os operadores de atribuição na tabela a seguir, exceto o `=` operador, são operadores de atribuição compostos.

Tabela de operadores de atribuição

OPERADOR	SIGNIFICADO
<code>=</code>	Armazena o valor do segundo operando no objeto especificado pelo primeiro operando (atribuição simples).
<code>*=</code>	Multiplica o valor do primeiro operando pelo valor do segundo operando; armazena o resultado no objeto especificado pelo primeiro operando.
<code>/=</code>	Divide o valor do primeiro operando pelo valor do segundo operando; armazena o resultado no objeto especificado pelo primeiro operando.
<code>%=</code>	Obtém o módulo do primeiro operando especificado pelo valor do segundo operando; armazena o resultado no objeto especificado pelo primeiro operando.
<code>+=</code>	Soma o valor do segundo operando ao valor do primeiro operando; armazena o resultado no objeto especificado pelo primeiro operando.
<code>-=</code>	Subtrai o valor do segundo operando do valor do primeiro operando; armazena o resultado no objeto especificado pelo primeiro operando.

OPERADOR	SIGNIFICADO
<code><<=</code>	Altera o valor do primeiro operando à esquerda do número de bits especificado pelo valor do segundo operando; armazena o resultado no objeto especificado pelo primeiro operando.
<code>>>=</code>	Altera o valor do primeiro operando à direita do número de bits especificado pelo valor do segundo operando; armazena o resultado no objeto especificado pelo primeiro operando.
<code>&=</code>	Obtém o bit a bit AND do primeiro e do segundo operandos; armazena o resultado no objeto especificado pelo primeiro operando.
<code>^=</code>	Obtém o bit a bit exclusivo OR do primeiro e do segundo operandos; armazena o resultado no objeto especificado pelo primeiro operando.
<code> =</code>	Obtém o bit a bit inclusivo OR do primeiro e do segundo operandos; armazena o resultado no objeto especificado pelo primeiro operando.

Palavras-chave do operador

Três dos operadores de atribuição compostos têm equivalentes de palavra-chave. Eles são:

OPERADOR	EQUIVALENTE
<code>&=</code>	<code>and_eq</code>
<code> =</code>	<code>or_eq</code>
<code>^=</code>	<code>xor_eq</code>

O C++ especifica essas palavras-chave de operador como grafias alternativas para os operadores de atribuição compostos. Em C, as grafias alternativas são fornecidas como macros no `<iso646.h>` cabeçalho. Em C++, as grafias alternativas são palavras-chave; o uso do `<iso646.h>` ou o equivalente em C++ `<ciso646>` é preterido. No Microsoft C++, a `/permissive-` opção ou do `/Za` compilador é necessária para habilitar a grafia alternativa.

Exemplo

```

// expre_Assignment_Operators.cpp
// compile with: /EHsc
// Demonstrate assignment operators
#include <iostream>
using namespace std;
int main() {
    int a = 3, b = 6, c = 10, d = 0xAAAA, e = 0x5555;

    a += b;      // a is 9
    b %= a;      // b is 6
    c >>= 1;     // c is 5
    d |= e;      // Bitwise--d is 0xFFFF

    cout << "a = 3, b = 6, c = 10, d = 0xAAAA, e = 0x5555" << endl
        << "a += b yields " << a << endl
        << "b %= a yields " << b << endl
        << "c >>= 1 yields " << c << endl
        << "d |= e yields " << hex << d << endl;
}

```

Atribuição simples

O operador de atribuição simples (`=`) faz com que o valor do segundo operando seja armazenado no objeto especificado pelo primeiro operando. Se ambos os objetos forem de tipos aritméticos, o operando à direita será convertido no tipo da esquerda, antes de armazenar o valor.

Os objetos `const` e `volatile` tipos podem ser atribuídos a valores l de tipos que são apenas `volatile` , ou que não são `const` ou `volatile` .

A atribuição a objetos do tipo de classe (`struct` , `union` e `class` tipos) é executada por uma função chamada `operator=` . O comportamento padrão dessa função do operador é executar uma cópia bit a bit; no entanto, esse comportamento pode ser alterado usando operadores sobrecarregados. Para obter mais informações, consulte [Sobrecarga de operador](#). Os tipos de classe também podem ter operadores de atribuição de *cópia* e de *movimentação* . Para obter mais informações, consulte [copiar construtores e copiar operadores de atribuição](#) e [mover construtores e mover operadores de atribuição](#).

Um objeto de qualquer classe derivada exclusiva de uma classe base pode ser atribuída a um objeto da classe base. O inverso não é verdadeiro porque há uma conversão implícita da classe derivada para a classe base, mas não da classe base para a classe derivada. Por exemplo:

```

// exre_SimpleAssignment.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;
class ABase
{
public:
    ABase() { cout << "constructing ABase\n"; }
};

class ADerived : public ABase
{
public:
    ADerived() { cout << "constructing ADerived\n"; }
};

int main()
{
    ABase aBase;
    ADerived aDerived;

    aBase = aDerived; // OK
    aDerived = aBase; // C2679
}

```

As atribuições para tipos de referência se comportam como se uma atribuição foi feita ao objeto ao qual a referência aponta.

Para objetos de tipo classe, a atribuição é diferente de inicialização. Para ilustrar como a atribuição e a inicialização podem ser diferentes, considere o código

```

UserType1 A;
UserType2 B = A;

```

O código anterior mostra um inicializador; ele chama o construtor de `UserType2` que usa um argumento do tipo `UserType1`. Dado o código

```

UserType1 A;
UserType2 B;

B = A;

```

a instrução de atribuição

```
B = A;
```

pode ter um dos seguintes efeitos

- Chame a função `operator=` para `UserType2`, fornecida `operator=` é fornecida com um `UserType1` argumento.
- Chama a função de conversão explícita `UserType1::operator UserType2`, se essa função existir.
- Chamar um construtor `UserType2::UserType2`, desde que essa construtor exista, que usa um argumento `UserType1` e copia o resultado.

Atribuição composta

Os operadores de atribuição compostos são mostrados na [tabela operadores de atribuição](#). Esses operadores têm a forma $E1 \ op = E2$, em que $E1$ é um `const` l-value não modificável e $E2$ é:

- um tipo aritmético
- um ponteiro, se op for `+` ou** `-`**

O formulário $E1 \ op = E2$ se comporta como $E1 \ = \ E1 \ op \ E2$, mas o $E1$ é avaliado apenas uma vez.

A atribuição composta para um tipo enumerado gera uma mensagem de erro. Se o operando esquerdo for de um tipo de ponteiro, o operando à direita deverá ser de um tipo de ponteiro ou deve ser uma expressão constante que seja avaliada como 0. Quando o operando esquerdo é de um tipo integral, o operando direito não deve ser de um tipo de ponteiro.

Resultado de operadores de atribuição

Os operadores de atribuição retornam o valor do objeto especificado pelo operando esquerdo após a atribuição. O tipo resultante é o tipo do operando esquerdo. O resultado de uma expressão de atribuição é sempre um l-value. Esses operadores binários possuem associatividade da direita para a esquerda. O operando esquerdo deve ser um l-value modificável.

No ANSI C, o resultado de uma expressão de atribuição não é um l-Value. Isso significa que a expressão de C++ legal `(a += b) += c` não é permitida em C.

Confira também

[Expressões com operadores binários](#)
[Operadores, precedência e Associação internos do C++](#)
[operadores de atribuição C](#)

Operador AND bit a bit:&

02/09/2020 • 2 minutes to read • [Edit Online](#)

Sintaxe

`expressão & de expressão de`

Comentários

O operador AND e AND () AND e Operator (`&`) compara cada bit do primeiro operando com o bit correspondente do segundo operando. Se ambos os bits forem 1, o bit de resultado correspondente será definido como 1. Caso contrário, o bit de resultado correspondente é definido como zero (0).

Ambos os operandos para o operador AND bit a bit devem ter tipos inteiros. As conversões aritméticas usuais abordadas nas [conversões padrão](#) são aplicadas aos operandos.

Palavra-chave Operator para &

O C++ especifica `bitand` como uma grafia alternativa para `&` . Em C, a grafia alternativa é fornecida como uma macro no `<iso646.h>` cabeçalho. Em C++, a grafia alternativa é uma palavra-chave; o uso do `<iso646.h>` ou o equivalente em C++ `<ciso646>` é preferido. No Microsoft C++, a `/permissive-` opção ou do `/za` compilador é necessária para habilitar a grafia alternativa.

Exemplo

```
// expre_Bitwise_AND_Operator.cpp
// compile with: /EHsc
// Demonstrate bitwise AND
#include <iostream>
using namespace std;
int main() {
    unsigned short a = 0xFFFF;      // pattern 1111 ...
    unsigned short b = 0xAAAA;      // pattern 1010 ...
    cout << hex << ( a & b ) << endl;  // prints "aaaa", pattern 1010 ...
}
```

Confira também

[Operadores, precedência e Associação internos do C++](#)

[Operadores bit a bit C](#)

Operador OR exclusivo bit a bit: ^

02/09/2020 • 2 minutes to read • [Edit Online](#)

Sintaxe

`expressão ^ de expressão de`

Comentários

O operador OR exclusivo OR bit (`^`) compara cada bit de seu primeiro operando com o bit correspondente de seu segundo operando. Se o bit em um dos operandos for 0 e o bit no outro operando for 1, o bit de resultado correspondente será definido como 1. Caso contrário, o bit de resultado correspondente é definido como zero (0).

Ambos os operandos para o operador devem ter tipos integrais. As conversões aritméticas usuais abordadas nas [conversões padrão](#) são aplicadas aos operandos.

Palavra-chave Operator para ^

O C++ especifica `xor` como uma grafia alternativa para `^`. Em C, a grafia alternativa é fornecida como uma macro no `<iso646.h>` cabeçalho. Em C++, a grafia alternativa é uma palavra-chave; o uso do `<iso646.h>` ou o equivalente em C++ `<ciso646>` é preferido. No Microsoft C++, a `/permissive-` opção ou do `/za` compilador é necessária para habilitar a grafia alternativa.

Exemplo

```
// expre_Bitwise_Exclusive_OR_Operator.cpp
// compile with: /EHsc
// Demonstrate bitwise exclusive OR
#include <iostream>
using namespace std;
int main() {
    unsigned short a = 0x5555;      // pattern 0101 ...
    unsigned short b = 0xFFFF;      // pattern 1111 ...

    cout << hex << ( a ^ b ) << endl;  // prints "aaaa" pattern 1010 ...
}
```

Confira também

[Operadores, precedência e Associação internos do C++](#)

Operador OR inclusivo de bits: |

02/09/2020 • 2 minutes to read • [Edit Online](#)

Sintaxe

`expression1 | expression2`

Comentários

O bit inclusivo OR (`|`) é comparado cada bit de seu primeiro operando com o bit correspondente de seu segundo operando. Se qualquer bit for 1, o bit de resultado correspondente será definido como 1. Caso contrário, o bit de resultado correspondente é definido como zero (0).

Ambos os operandos para o operador devem ter tipos integrais. As conversões aritméticas usuais abordadas nas [conversões padrão](#) são aplicadas aos operandos.

Palavra-chave Operator para |

O C++ especifica `bitor` como uma grafia alternativa para `|`. Em C, a grafia alternativa é fornecida como uma macro no `<iso646.h>` cabeçalho. Em C++, a grafia alternativa é uma palavra-chave; o uso do `<iso646.h>` ou o equivalente em C++ `<ciso646>` é preferido. No Microsoft C++, a `/permissive-` opção ou do `/za` compilador é necessária para habilitar a grafia alternativa.

Exemplo

```
// expre_Bitwise_Inclusive_OR_Operator.cpp
// compile with: /EHsc
// Demonstrate bitwise inclusive OR
#include <iostream>
using namespace std;

int main() {
    unsigned short a = 0x5555;      // pattern 0101 ...
    unsigned short b = 0xAAAA;      // pattern 1010 ...

    cout << hex << ( a | b ) << endl;  // prints "ffff" pattern 1111 ...
}
```

Confira também

[Operadores, precedência e Associação internos do C++](#)

[Operadores bit a bit C](#)

Operador cast: ()

25/03/2020 • 2 minutes to read • [Edit Online](#)

Uma conversão de tipo fornece um método para conversão explícita do tipo de um objeto em uma situação específica.

Sintaxe

```
unary-expression ( type-name ) cast-expression
```

Comentários

Qualquer expressão unária é considerada uma expressão de conversão.

O compilador trata *cast-expression* como tipo *type-name* depois que uma conversão de tipo é feita. As conversões podem ser usadas para converter objetos de qualquer tipo escalar em ou de qualquer outro tipo escalar. As conversões de tipo explícito são restrinvidas pelas mesmas regras que determinam os efeitos de conversões implícitas. As restrições adicionais de conversões podem resultar de tamanhos reais ou de representação de tipos específicos.

Exemplo

```
// expre_CastOperator.cpp
// compile with: /EHsc
// Demonstrate cast operator
#include <iostream>

using namespace std;

int main()
{
    double x = 3.1;
    int i;
    cout << "x = " << x << endl;
    i = (int)x;    // assign i the integer part of x
    cout << "i = " << i << endl;
}
```

Exemplo

```

// expre_CastOperator2.cpp
// The following sample shows how to define and use a cast operator.
#include <string.h>
#include <stdio.h>

class CountedAnsiString
{
public:
    // Assume source is not null terminated
    CountedAnsiString(const char *pStr, size_t nSize) :
        m_nSize(nSize)
    {
        m_pStr = new char[sizeOfBuffer];

        strncpy_s(m_pStr, sizeOfBuffer, pStr, m_nSize);
        memset(&m_pStr[m_nSize], '!', 9); // for demonstration purposes.
    }

    // Various string-like methods...

    const char *GetRawBytes() const
    {
        return(m_pStr);
    }

    //
    // operator to cast to a const char *
    //
    operator const char *()
    {
        m_pStr[m_nSize] = '\0';
        return(m_pStr);
    }

    enum
    {
        sizeOfBuffer = 20
    } size;

private:
    char *m_pStr;
    const size_t m_nSize;
};

int main()
{
    const char *kStr = "Excitinggg";
    CountedAnsiString myStr(kStr, 8);

    const char *pRaw = myStr.GetRawBytes();
    printf_s("RawBytes truncated to 10 chars: %.10s\n", pRaw);

    const char *pCast = myStr; // or (const char *)myStr;
    printf_s("Casted Bytes: %s\n", pCast);

    puts("Note that the cast changed the raw internal string");
    printf_s("Raw Bytes after cast: %s\n", pRaw);
}

```

```

RawBytes truncated to 10 chars: Exciting!!
Casted Bytes: Exciting
Note that the cast changed the raw internal string
Raw Bytes after cast: Exciting

```

Confira também

[Expressões com operadores unários](#)

[Operadores internos, precedência e associatividade C++](#)

[Operador de conversão de tipo explícito: \(\)](#)

[Operadores de conversão](#)

[Operadores cast](#)

Operador de vírgula: ,

25/03/2020 • 2 minutes to read • [Edit Online](#)

Permite agrupar duas instruções onde uma é esperado.

Sintaxe

```
expression , expression
```

Comentários

O operador vírgula tem associatividade da esquerda para a direita. Duas expressões separadas por uma vírgula são avaliadas da esquerda para a direita. O operando à esquerda é sempre avaliado, e todos os efeitos colaterais são concluídos antes de o operando à direita ser avaliado.

As vírgulas podem ser usadas como separadores em alguns contextos, como listas de argumentos de função. Não confunda o uso da vírgula como separador com seu uso como operador; os dois usos são completamente diferentes.

Considere a expressão `e1, e2`. O tipo e o valor da expressão são o tipo e o valor de $E2$; o resultado da avaliação do $E1$ é Descartado. O resultado será um valor I se o operando à direita for um valor I .

Onde a vírgula é geralmente usada como separador (por exemplo, nos argumentos reais para funções ou inicializadores agregados), o operador vírgula e seus operandos devem ser colocados entre parênteses. Por exemplo:

```
func_one( x, y + 2, z );
func_two( (x--, y + 2), z );
```

Na chamada de função para `func_one` acima, três argumentos, separados por vírgulas, são passados: `x`, `y + 2` e `z`. Na chamada da função para `func_two`, os parênteses forçam o compilador a interpretar a primeira vírgula como o operador de avaliação sequencial. Essa chamada de função passa dois argumentos para `func_two`. O primeiro argumento é o resultado da operação de avaliação sequencial `(x--, y + 2)`, que tem o valor e o tipo da expressão `y + 2`; o segundo argumento é `z`.

Exemplo

```
// cpp_comma_operator.cpp
#include <stdio.h>
int main () {
    int i = 10, b = 20, c = 30;
    i = b, c;
    printf("%i\n", i);

    i = (b, c);
    printf("%i\n", i);
}
```

Confira também

[Expressões com operadores binários](#)

[Operadores internos, precedência e associatividade C++](#)

[Operador de avaliação sequencial](#)

Operador condicional: ?: :

02/09/2020 • 4 minutes to read • [Edit Online](#)

Sintaxe

```
expression ? expression : expression
```

Comentários

O operador condicional (?:) é um operador ternário (ele usa três operandos). O operador condicional funciona desta forma:

- O primeiro operando é implicitamente convertido em `bool`. É avaliado e todos os efeitos colaterais são concluídos antes de continuar.
- Se o primeiro operando for avaliado como `true` (1), o segundo operando será avaliado.
- Se o primeiro operando for avaliado como `false` (0), o terceiro operando será avaliado.

O resultado do operador condicional é o resultado de qualquer operando avaliado — o segundo ou o terceiro. Somente um dos dois operandos dos dois mais recentes é avaliado em uma expressão condicional.

As expressões condicionais têm a capacidade da direita para a esquerda. O primeiro operando deve ser do tipo integral ou ponteiro. As regras a seguir se aplicam ao segundo e ao terceiro operando:

- Se ambos os operandos forem do mesmo tipo, o resultado será desse tipo.
- Se ambos os operandos forem de tipos aritméticos ou de enumeração, as conversões aritméticas usuais (cobertas em [conversões padrão](#)) serão executadas para convertê-los em um tipo comum.
- Se ambos os operandos forem de tipos de ponteiro ou se um for um tipo de ponteiro e o outro for uma expressão constante que é avaliada como 0, as conversões de ponteiro serão executadas para convertê-las em um tipo comum.
- Se ambos os operandos forem de tipos de referência, as conversões de referência serão executadas para convertê-los em um tipo comum.
- Se ambos os operandos forem do tipo `void`, o tipo comum será de tipo `void`.
- Se ambos os operandos forem do mesmo tipo definido pelo usuário, o tipo comum será esse tipo.
- Se os operandos tiverem tipos diferentes e pelo menos um dos operandos tiver o tipo definido pelo usuário, as regras de idioma serão usadas para determinar o tipo comum. (Consulte o aviso abaixo.)

Todas as combinações do segundo e do terceiro operando que não estiverem na lista anterior são ilegais. O tipo de resultado é o tipo comum, e é um l-value se o segundo e o terceiro operandos forem do mesmo tipo e ambos forem l-values.

WARNING

Se os tipos do segundo e do terceiro operandos não forem idênticos, as regras de conversão de tipo complexo, conforme especificado no padrão C++, serão invocadas. Essas conversões podem levar a um comportamento inesperado, incluindo a construção e a destruição de objetos temporários. Por esse motivo, Aconselhamos enfaticamente que você (1) Evite usar tipos definidos pelo usuário como operandos com o operador condicional ou (2) se você usar tipos definidos pelo usuário e, em seguida, converter explicitamente cada operando em um tipo comum.

Exemplo

```
// expre_Expressions_with_the_Conditional_Operator.cpp
// compile with: /EHsc
// Demonstrate conditional operator
#include <iostream>
using namespace std;
int main() {
    int i = 1, j = 2;
    cout << ( i > j ? i : j ) << " is greater." << endl;
}
```

Confira também

[Operadores internos C++, precedência e associatividade](#)

[Operador de expressão condicional](#)

Operador delete (C++)

02/09/2020 • 5 minutes to read • [Edit Online](#)

Desaloca um bloco de memória.

Sintaxe

```
[ :: ] delete expressão de conversão  
[ :: ] delete [] expressão de conversão
```

Comentários

O argumento *Cast-Expression* deve ser um ponteiro para um bloco de memória alocada anteriormente para um objeto criado com o [operador New](#). O `delete` operador tem um resultado do tipo `void` e, portanto, não retorna um valor. Por exemplo:

```
CDialog* MyDialog = new CDialog;  
// use MyDialog  
delete MyDialog;
```

O uso `delete` de um ponteiro para um objeto não alocado com o `new` fornece resultados imprevisíveis. No entanto, você pode usar `delete` em um ponteiro com o valor 0. Essa provisão significa que, quando `new` retorna 0 em caso de falha, a exclusão do resultado de uma operação com falha `new` é inofensiva. Para obter mais informações, consulte [os operadores New e Delete](#).

Os `new` `delete` operadores e também podem ser usados para tipos internos, incluindo matrizes. Se se `pointer` referir a uma matriz, coloque colchetes vazios (`[]`) antes de `pointer` :

```
int* set = new int[100];  
//use set[]  
delete [] set;
```

O uso do `delete` operador em um objeto desaloca sua memória. Um programa que remova a referência de um ponteiro após a exclusão do objeto pode ter resultados imprevisíveis ou falhar.

Quando `delete` é usado para desalocar memória para um objeto de classe C++, o destruidor do objeto é chamado antes de a memória do objeto ser desalocada (se o objeto tiver um destruidor).

Se o operando para o `delete` operador for um valor l modificado, seu valor será indefinido depois que o objeto for excluído.

Se a opção de compilador [/SDL \(habilitar verificações de segurança adicionais\)](#) for especificada, o operando para o `delete` operador será definido como um valor inválido depois que o objeto for excluído.

Usando delete

Há duas variantes sintáticas para o [operador Delete](#): uma para objetos únicos e a outra para matrizes de objetos. O fragmento de código a seguir mostra como diferem:

```

// expre_Using_delete.cpp
struct UDTtype
{
};

int main()
{
    // Allocate a user-defined object, UDOobject, and an object
    // of type double on the free store using the
    // new operator.
    UDTtype *UDOobject = new UDTtype;
    double *dObject = new double;
    // Delete the two objects.
    delete UDOobject;
    delete dObject;
    // Allocate an array of user-defined objects on the
    // free store using the new operator.
    UDTtype (*UDArr)[7] = new UDTtype[5][7];
    // Use the array syntax to delete the array of objects.
    delete [] UDArr;
}

```

Os dois casos a seguir produzem resultados indefinidos: usando a forma de matriz de Delete (`delete []`) em um objeto e usando a forma nonarray de Delete em uma matriz.

Exemplo

Para obter exemplos de como usar `delete` , consulte [novo operador](#).

Como excluir funciona

O operador Delete invoca a função de **exclusão do operador**.

Para objetos que não são do tipo de classe ([Class, struct ou Union](#)), o operador global Delete é invocado. Para objetos do tipo de classe, o nome da função de desalocação será resolvido no escopo global se a expressão de exclusão começar com o operador unário de resolução de escopo (`::`). Caso contrário, o operador Delete invoca o destruidor para um objeto antes de desalocar a memória (se o ponteiro não for nulo). O operador delete pode ser definido em uma base por classe; se não houver nenhuma definição para uma classe específica, a exclusão global do operador é chamada. Se a expressão de exclusão for usada para desalocar um objeto da classe cujo tipo estático tem um destruidor virtual, a função de desalocação é resolvida pelo destruidor virtual do tipo dinâmico do objeto.

Confira também

[Expressões com operadores unários](#)

[Palavras-chave](#)

[Operadores New e Delete](#)

Operadores de igualdade: == e !=

02/09/2020 • 2 minutes to read • [Edit Online](#)

Sintaxe

<i>expressão</i>	<code>==</code>	de <i>expressão</i> de
<i>expressão</i>	<code>!=</code>	de <i>expressão</i> de

Comentários

Operadores de igualdade binários compararão seus operandos em busca de igualdades ou desigualdades estritas.

Os operadores de igualdade, igual a (`==`) e diferente de (`!=`), têm precedência mais baixa do que os operadores relacionais, mas se comportam de forma semelhante. O tipo de resultado para esses operadores é `bool`.

O operador equal-to (`==`) retornará `true` se ambos os operandos tiverem o mesmo valor; caso contrário, retornará `false`. O operador NOT-equal-to (`!=`) retorna `true` se os operandos não têm o mesmo valor; caso contrário, ele retorna `false`.

Palavra-chave Operator para! =

O C++ especifica `not_eq` como uma grafia alternativa para `!=`. (Não há nenhuma grafia alternativa para `==`.) Em C, a grafia alternativa é fornecida como uma macro no `<iso646.h>` cabeçalho. Em C++, a grafia alternativa é uma palavra-chave; o uso do `<iso646.h>` ou o equivalente em C++ `<ciso646>` é preferido. No Microsoft C++, a `/permissive-` opção ou do `/Za` compilador é necessária para habilitar a grafia alternativa.

Exemplo

```
// expe_Equality_Operators.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;

int main() {
    cout << boolalpha
        << "The true expression 3 != 2 yields: "
        << (3 != 2) << endl
        << "The false expression 20 == 10 yields: "
        << (20 == 10) << endl;
}
```

Os operadores de igualdade podem comparar ponteiros a membros do mesmo tipo. Nessa comparação, as conversões de ponteiro para membro são executadas. Os ponteiros para membros também podem ser comparados a uma expressão constante que é avaliada como 0.

Confira também

[Expressões com operadores binários](#)

[Operadores internos do C++, precedência; e Associação](#)

Operador de conversão de tipo explícito: ()

15/04/2020 • 3 minutes to read • [Edit Online](#)

O C++ permite a conversão de tipos explícita usando uma sintaxe semelhante à sintaxe de chamada de função.

Sintaxe

```
simple-type-name ( expression-list )
```

Comentários

Um *nome de tipo simples* seguido de uma lista de *expressões* incluída entre parênteses constrói um objeto do tipo especificado usando as expressões especificadas. O exemplo a seguir mostra uma conversão de tipo explícita para o tipo int:

```
int i = int( d );
```

O exemplo a [Point](#) seguir mostra uma classe.

Exemplo

```

// expre_Explicit_Type_Conversion_Operator.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
class Point
{
public:
    // Define default constructor.
    Point() { _x = _y = 0; }
    // Define another constructor.
    Point( int X, int Y ) { _x = X; _y = Y; }

    // Define "accessor" functions as
    // reference types.
    unsigned& x() { return _x; }
    unsigned& y() { return _y; }
    void Show() { cout << "x = " << _x << ", "
                  << "y = " << _y << "\n"; }

private:
    unsigned _x;
    unsigned _y;
};

int main()
{
    Point Point1, Point2;

    // Assign Point1 the explicit conversion
    // of ( 10, 10 ).
    Point1 = Point( 10, 10 );

    // Use x() as an l-value by assigning an explicit
    // conversion of 20 to type unsigned.
    Point1.x() = unsigned( 20 );
    Point1.Show();

    // Assign Point2 the default Point object.
    Point2 = Point();
    Point2.Show();
}

```

Saída

```

x = 20, y = 10
x = 0, y = 0

```

Embora o exemplo acima demonstre a conversão de tipos explícita usando constantes, a mesma técnica funciona para executar essas conversões em objetos. O fragmento de código a seguir demonstra isso:

```

int i = 7;
float d;

d = float( i );

```

As conversões de tipos explícitas também podem ser especificadas usando a sintaxe em estilo "cast". O exemplo anterior, reescrito usando a sintaxe "cast", fica assim:

```

d = (float)i;

```

As conversões em estilo "cast" e em estilo de função têm os mesmos resultados na conversão de valores únicos. Porém, na sintaxe em estilo de função, você pode especificar mais de um argumento para a conversão. Essa diferença é importante para os tipos definidos pelo usuário. Considere uma classe `Point` e as respectivas conversões:

```
struct Point
{
    Point( short x, short y ) { _x = x; _y = y; }
    ...
    short _x, _y;
};

...
Point pt = Point( 3, 10 );
```

O exemplo anterior, que usa a conversão no estilo de função, mostra como converter dois `Point` valores (um para `x` e um para `y`) para o tipo definido pelo usuário .

Caution

Use as conversões de tipos explícitas com cuidado, já que elas substituem a verificação de tipos interna do compilador do C++.

A notação [do elenco](#) deve ser usada para conversões para tipos que não tenham um nome de tipo simples (ponteiro ou tipos *de* referência, por exemplo). A conversão para tipos que podem ser expressos com um *nome de tipo simple* pode ser escrita em qualquer forma.

A definição de tipo nas conversões em estilo "cast" é ilegal.

Confira também

[Expressões pós-fixadas](#)

[Operadores internos C++, precedência e associatividade](#)

Operador de chamada da função: ()

02/09/2020 • 6 minutes to read • [Edit Online](#)

Uma chamada de função é um tipo de `postfix-expression`, formada por uma expressão que é avaliada como uma função ou um objeto que possa ser chamado, seguido pelo operador de chamada de função, `()`. Um objeto pode declarar uma `operator ()` função, que fornece semântica de chamada de função para o objeto.

Syntax

```
postfix-expression :  
    postfix-expression ( argument-expression-List ) opt )
```

Comentários

Os argumentos para o operador de chamada de função são provenientes de uma `argument-expression-List` lista de expressões separadas por vírgulas. Os valores dessas expressões são passados para a função como argumentos. A *lista de expressão de argumento* pode estar vazia. Antes do C++ 17, a ordem de avaliação da expressão de função e as expressões de argumento não são especificadas e podem ocorrer em qualquer ordem. No C++ 17 e posteriores, a expressão de função é avaliada antes de qualquer expressão de argumento ou argumentos padrão. As expressões de argumento são avaliadas em uma sequência indeterminada.

O `postfix-expression` é avaliado para a função a ser chamada. Ele pode ter várias formas:

- um identificador de função, visível no escopo atual ou no escopo de qualquer um dos argumentos de função fornecidos,
- uma expressão que é avaliada como uma função, um ponteiro de função, um objeto que possa ser chamado ou uma referência a uma,
- um acessador de função membro, explícito ou implícito,
- um ponteiro de referência para uma função de membro.

O `postfix-expression` pode ser um identificador de função sobrecarregado ou um acessador de função de membro sobrecarregado. As regras para resolução de sobrecarga determinam a função real a ser chamada. Se a função de membro for virtual, a função a ser chamada será determinada em tempo de execução.

Algumas declarações de exemplo:

- Tipo de retorno de função `T`. Uma declaração de exemplo é

```
T func( int i );
```

- Tipo de retorno de ponteiro para uma função `T`. Uma declaração de exemplo é

```
T (*func)( int i );
```

- Tipo de retorno de referência para uma função `T`. Uma declaração de exemplo é

```
T (&func)(int i );
```

- Tipo de retorno de desreferência de ponteiro para função de membro `T`. As chamadas de função de exemplo são

```
(pObject->*pmf)();  
(Object.*pmf)();
```

Exemplo

O exemplo a seguir chama a função de biblioteca padrão `strcat_s` com três argumentos:

```
// expre_Function_Call_Operator.cpp  
// compile with: /EHsc  
  
#include <iostream>  
#include <string>  
  
// C++ Standard Library name space  
using namespace std;  
  
int main()  
{  
    enum  
    {  
        sizeOfBuffer = 20  
    };  
  
    char s1[ sizeOfBuffer ] = "Welcome to ";  
    char s2[ ] = "C++";  
  
    strcat_s( s1, sizeOfBuffer, s2 );  
  
    cout << s1 << endl;  
}
```

```
Welcome to C++
```

Resultados da chamada de função

Uma chamada de função é avaliada como um Rvalue, a menos que a função seja declarada como um tipo de referência. As funções com tipos de retorno de referência são avaliadas como lvalue. Essas funções podem ser usadas no lado esquerdo de uma instrução de atribuição, como visto aqui:

```

// expre_Function_Call_Results.cpp
// compile with: /EHsc
#include <iostream>
class Point
{
public:
    // Define "accessor" functions as
    // reference types.
    unsigned& x() { return _x; }
    unsigned& y() { return _y; }
private:
    unsigned _x;
    unsigned _y;
};

using namespace std;
int main()
{
    Point ThePoint;

    ThePoint.x() = 7;           // Use x() as an l-value.
    unsigned y = ThePoint.y(); // Use y() as an r-value.

    // Use x() and y() as r-values.
    cout << "x = " << ThePoint.x() << "\n"
        << "y = " << ThePoint.y() << "\n";
}

```

O código anterior define uma classe chamada `Point`, que contém objetos de dados privados que representam coordenadas `x` e `y`. Esses objetos de dados devem ser alterados e seus valores recuperados. Esse programa é apenas um de vários projetos para essa classe; o uso das funções `GetX` e `SetX` ou `GetY` e `SetY` é outro projeto possível.

As funções que retornam tipos de classe, os ponteiros para tipos de classe ou referências a tipos de classe podem ser usados como o operando à esquerda para operadores de seleção de membros. O código a seguir é legal:

```

// expre_Function_Results2.cpp
class A {
public:
    A() {}
    A(int i) {}
    int SetA( int i ) {
        return (I = i);
    }

    int GetA() {
        return I;
    }

private:
    int I;
};

A func1() {
    A a = 0;
    return a;
}

A* func2() {
    A *a = new A();
    return a;
}

A& func3() {
    A *a = new A();
    A &b = *a;
    return b;
}

int main() {
    int iResult = func1().GetA();
    func2()->SetA( 3 );
    func3().SetA( 7 );
}

```

As funções podem ser chamadas recursivamente. Para obter mais informações sobre declarações de função, consulte [funções](#). O material relacionado está em [unidades de tradução e vinculação](#).

Veja também

[Expressões pós-fixadas](#)

[Operadores, precedência e Associação internos do C++](#)

[Chamada de função](#)

Operador de indireção: *

25/03/2020 • 2 minutes to read • [Edit Online](#)

Sintaxe

```
* cast-expression
```

Comentários

O operador de indireção unário (*) faz referência a um ponteiro; ou seja, ele converte um valor de ponteiro para um l-Value. O operando do operador de indireção deve ser um ponteiro para um tipo. O resultado da expressão de indireção é o tipo do qual o tipo do ponteiro é derivado. O uso do operador de * nesse contexto é diferente do seu significado como um operador binário, que é a multiplicação.

Se o operando apontar para uma função, o resultado será um designador de função. Se ele apontar para um local de armazenamento, o resultado será um valor l que designa o local de armazenamento.

O operador de indireção pode ser usado cumulativamente para retirar as referências dos ponteiros para os ponteiros. Por exemplo:

```
// expre_Indirection_Operator.cpp
// compile with: /EHsc
// Demonstrate indirection operator
#include <iostream>
using namespace std;
int main()
{
    int n = 5;
    int *pn = &n;
    int **ppn = &pn;

    cout << "Value of n:\n"
        << "direct value: " << n << endl
        << "indirect value: " << *pn << endl
        << "doubly indirect value: " << **ppn << endl
        << "address of n: " << pn << endl
        << "address of n via indirection: " << *ppn << endl;
}
```

Se o valor do ponteiro for inválido, o resultado será indefinido. A lista a seguir inclui algumas das condições mais comuns que invalidam um valor de ponteiro.

- O ponteiro é um ponteiro nulo.
- O ponteiro especifica o endereço de um item local que não está visível no momento da referência.
- O ponteiro especifica um endereço que está alinhado de forma inadequada para o tipo do objeto apontado.
- O ponteiro especifica um endereço não usado pelo programa em execução.

Confira também

[Expressões com operadores unários](#)

[Operadores internos, precedência e associatividade C++](#)

[Operador endereço de: &](#)

Operadores SHIFT esquerda e SHIFT direita (>> e <<)

02/09/2020 • 9 minutes to read • [Edit Online](#)

Os operadores de alternância de bits bit a passo são o operador de deslocamento à >> direita (), que move o bit da *expressão Shift* para o lado direito e o operador esquerdo-SHIFT (<<), que move os bits da *expressão Shift* para a esquerda.¹

Sintaxe

```
expressão Shift << expressão aditiva
shift-expression >> additive-expression
```

Comentários

IMPORTANT

As seguintes descrições e exemplos são válidos no Windows para arquiteturas x86 e x64. A implementação dos operadores de deslocamento para a esquerda e para a direita é significativamente diferente no Windows para dispositivos ARM. Para obter mais informações, consulte a seção "mover operadores" da postagem no blog do [Hello ARM](#) .

Deslocamentos para a esquerda

O operador Left-Shift faz com que os bits na *expressão Shift* sejam deslocados para a esquerda pelo número de posições especificadas pela *expressão aditiva*. As posições de bits que foram liberadas pela operação de deslocamento são preenchidas com zeros. Um deslocamento para a esquerda é um deslocamento lógico (os bits que são deslocados da extremidade são descartados, incluindo o bit de sinal). Para obter mais informações sobre os tipos de turnos de bits, consulte [turnos de bits](#).

O exemplo a seguir mostra operações de deslocamento para a esquerda usando números sem sinal. O exemplo a seguir mostra o que está acontecendo com os bit representando o valor como um conjunto de bits. Para obter mais informações, consulte [classe conjunto](#).

```

#include <iostream>
#include <bitset>

using namespace std;

int main() {
    unsigned short short1 = 4;
    bitset<16> bitset1{short1};    // the bitset representation of 4
    cout << bitset1 << endl;    // 0b00000000'0000100

    unsigned short short2 = short1 << 1;      // 4 left-shifted by 1 = 8
    bitset<16> bitset2{short2};
    cout << bitset2 << endl;    // 0b00000000'00001000

    unsigned short short3 = short1 << 2;      // 4 left-shifted by 2 = 16
    bitset<16> bitset3{short3};
    cout << bitset3 << endl;    // 0b00000000'00010000
}

```

Se você deslocar um número com sinal para a esquerda de forma que o bit de sinal seja afetado, o resultado será indefinido. O exemplo a seguir mostra o que acontece quando um 1 bit é deslocado para a posição do bit de sinal.

```

#include <iostream>
#include <bitset>

using namespace std;

int main() {
    short short1 = 16384;
    bitset<16> bitset1(short1);
    cout << bitset1 << endl;    // 0b01000000'00000000

    short short3 = short1 << 1;
    bitset<16> bitset3(short3);    // 16384 left-shifted by 1 = -32768
    cout << bitset3 << endl;    // 0b10000000'00000000

    short short4 = short1 << 14;
    bitset<16> bitset4(short4);    // 4 left-shifted by 14 = 0
    cout << bitset4 << endl;    // 0b00000000'00000000
}

```

Deslocamentos para a direita

O operador right-shift faz com que o padrão de bit na *expressão Shift* seja deslocado para a direita pelo número de posições especificadas pela *expressão aditiva*. Para números sem sinal, as posições de bits que foram liberadas pela operação de deslocamento são preenchidas com zeros. Para números com sinal, o bit de sinal é usado para preencher as posições de bit vagas. Ou seja, se o número for positivo, 0 será usado, e se o número for negativo, 1 será usado.

IMPORTANT

O resultado do deslocamento de um número negativo para a direita dependerá da implementação. Embora o compilador do Microsoft C++ use o bit de sinal para preencher as posições de bits vagas, não há nenhuma garantia de que outras implementações também façam isso.

Este exemplo mostra operações de deslocamento para a direita usando números sem sinal:

```

#include <iostream>
#include <bitset>

using namespace std;

int main() {
    unsigned short short11 = 1024;
    bitset<16> bitset11{short11};
    cout << bitset11 << endl;      // 0b00000100'00000000

    unsigned short short12 = short11 >> 1; // 512
    bitset<16> bitset12{short12};
    cout << bitset12 << endl;      // 0b00000010'00000000

    unsigned short short13 = short11 >> 10; // 1
    bitset<16> bitset13{short13};
    cout << bitset13 << endl;      // 0b00000000'00000001

    unsigned short short14 = short11 >> 11; // 0
    bitset<16> bitset14{short14};
    cout << bitset14 << endl;      // 0b00000000'00000000
}

```

O próximo exemplo mostra operações de deslocamento para a direita com números positivos com sinal.

```

#include <iostream>
#include <bitset>

using namespace std;

int main() {
    short short1 = 1024;
    bitset<16> bitset1(short1);
    cout << bitset1 << endl;      // 0b00000100'00000000

    short short2 = short1 >> 1; // 512
    bitset<16> bitset2(short2);
    cout << bitset2 << endl;      // 0b00000010'00000000

    short short3 = short1 >> 11; // 0
    bitset<16> bitset3(short3);
    cout << bitset3 << endl;      // 0b00000000'00000000
}

```

O próximo exemplo mostra operações de deslocamento para a direita com inteiros negativos com sinal.

```

#include <iostream>
#include <bitset>

using namespace std;

int main() {
    short neg1 = -16;
    bitset<16> bn1(neg1);
    cout << bn1 << endl; // 0b11111111'11110000

    short neg2 = neg1 >> 1; // -8
    bitset<16> bn2(neg2);
    cout << bn2 << endl; // 0b11111111'11111000

    short neg3 = neg1 >> 2; // -4
    bitset<16> bn3(neg3);
    cout << bn3 << endl; // 0b11111111'11111100

    short neg4 = neg1 >> 4; // -1
    bitset<16> bn4(neg4);
    cout << bn4 << endl; // 0b11111111'11111111

    short neg5 = neg1 >> 5; // -1
    bitset<16> bn5(neg5);
    cout << bn5 << endl; // 0b11111111'11111111
}

```

Deslocamentos e promoções

As expressões em ambos os lados do operador shift devem ser tipos integrais. As promoções integrais são executadas de acordo com as regras descritas nas [conversões padrão](#). O tipo do resultado é o mesmo que o tipo da *expressão Shift* promovida.

No exemplo a seguir, uma variável do tipo `char` é promovida para um `int`.

```

#include <iostream>
#include <typeinfo>

using namespace std;

int main() {
    char char1 = 'a';

    auto promoted1 = char1 << 1; // 194
    cout << typeid(promoted1).name() << endl; // int

    auto promoted2 = char1 << 10; // 99328
    cout << typeid(promoted2).name() << endl; // int
}

```

Additional Details

O resultado de uma operação de deslocamento será indefinido se a *expressão aditiva* for negativa ou se a *expressão aditiva* for maior ou igual ao número de bits na *expressão Shift* (promovida). Nenhuma operação de deslocamento será executada se a *expressão aditiva* for 0.

```

#include <iostream>
#include <bitset>

using namespace std;

int main() {
    unsigned int int1 = 4;
    bitset<32> b1{int1};
    cout << b1 << endl;    // 0b00000000'00000000'00000000'00000100

    unsigned int int2 = int1 << -3; // C4293: '<<' : shift count negative or too big, undefined behavior
    unsigned int int3 = int1 >> -3; // C4293: '>>' : shift count negative or too big, undefined behavior
    unsigned int int4 = int1 << 32; // C4293: '<<' : shift count negative or too big, undefined behavior
    unsigned int int5 = int1 >> 32; // C4293: '>>' : shift count negative or too big, undefined behavior
    unsigned int int6 = int1 << 0;
    bitset<32> b6{int6};
    cout << b6 << endl;    // 0b00000000'00000000'00000000'00000100 (no change)
}

```

Notas de rodapé

¹ a seguir está a descrição dos operadores de mudança na Especificação ISO 11 do C++ (INCITS/ISO/IEC 14882-2011 [2012]), Sections 5.8.2 e 5.8.3.

O valor de $E1 \ll E2$ é $E1$ deslocado para a esquerda nas posições do bit $E2$; os bits vagos são preenchidos por zero. Se $E1$ tiver um tipo não assinado, o valor do resultado será $E1 \times 2^{E2}$, módulo reduzido um mais do que o valor máximo representável no tipo de resultado. Caso contrário, se $E1$ o tiver um tipo assinado e um valor não negativo, e $E1 \times 2^{E2}$ for representável no tipo não assinado correspondente do tipo de resultado, esse valor, convertido no tipo de resultado, será o valor resultante; caso contrário, o comportamento será indefinido.

O valor de $E1 \gg E2$ é $E1$ é deslocado para a direita nas posições do bit $E2$. Se o $E1$ tiver um tipo não assinado ou se $E1$ tiver um tipo assinado e um valor não negativo, o valor do resultado será a parte integral do quociente de $E1/2^{E2}$. Se $E1$ tiver um tipo com sinal e um valor negativo, o valor resultante será definido pela implementação.

Confira também

[Expressões com operadores binários](#)

[Operadores internos C++, precedência e associatividade](#)

Operador AND lógico: &&

02/09/2020 • 2 minutes to read • [Edit Online](#)

Sintaxe

```
expressão && de expressão de
```

Comentários

O operador AND lógico (`&&`) retorna `true` se ambos os operandos são `true` e retorna de `false` outra forma. Os operandos são convertidos implicitamente em tipo `bool` antes da avaliação e o resultado é do tipo `bool` . O AND lógico tem associatividade da esquerda para a direita.

Os operandos para o operador AND lógico não precisam ter o mesmo tipo, mas devem ter o tipo booleano, integral ou de ponteiro. Os operandos são geralmente expressões relacionais ou de igualdade.

O primeiro operando é completamente avaliado e todos os efeitos colaterais são concluídos antes de a avaliação da expressão AND lógica continuar.

O segundo operando será avaliado somente se o primeiro operando for avaliado como `true` (diferente de zero). Essa avaliação elimina a avaliação desnecessária do segundo operando quando a expressão AND lógica é `false` . Você pode usar a avaliação de curto-circuito para evitar a remoção de referência do ponteiro nulo, conforme mostrado no seguinte exemplo:

```
char *pch = 0;  
// ...  
(pch) && (*pch = 'a');
```

Se `pch` for nulo (0), o lado direito da expressão nunca será avaliado. Essa avaliação de curto-circuito torna impossível a atribuição por meio de um ponteiro nulo.

Palavra-chave Operator para &&

O C++ especifica `and` como uma grafia alternativa para `&&` . Em C, a grafia alternativa é fornecida como uma macro no `<iso646.h>` cabeçalho. Em C++, a grafia alternativa é uma palavra-chave; o uso do `<iso646.h>` ou o equivalente em C++ `<ciso646>` é preferido. No Microsoft C++, a `/permissive-` opção ou do `/za` compilador é necessária para habilitar a grafia alternativa.

Exemplo

```
// expre_Logical_AND_Operator.cpp
// compile with: /EHsc
// Demonstrate logical AND
#include <iostream>

using namespace std;

int main() {
    int a = 5, b = 10, c = 15;
    cout << boolalpha
        << "The true expression "
        << "a < b && b < c yields "
        << (a < b && b < c) << endl
        << "The false expression "
        << "a > b && b < c yields "
        << (a > b && b < c) << endl;
}
```

Confira também

[Operadores, precedência e Associação internos do C++](#)

[Operadores lógicos C](#)

Operador de negação lógico: !

02/09/2020 • 2 minutes to read • [Edit Online](#)

Sintaxe

```
* ! ***expressão CAST
```

Comentários

O operador lógico de negação (`!`) inverte o significado de seu operando. O operando deve ser do tipo aritmético ou ponteiro (ou uma expressão que é avaliada para o tipo aritmético ou ponteiro). O operando é implicitamente convertido para o tipo `bool` . O resultado será `true` se o operando convertido for `false` ; o resultado será `false` se o operando convertido for `true` . O resultado é do tipo `bool` .

Para uma expressão `e` , a expressão unário `!e` é equivalente à expressão `(e == 0)` , exceto onde operadores sobrecarregados estão envolvidos.

Palavra-chave Operator para!

O C++ especifica `not` como uma grafia alternativa para `!` . Em C, a grafia alternativa é fornecida como uma macro no `<iso646.h>` cabeçalho. Em C++, a grafia alternativa é uma palavra-chave; o uso do `<iso646.h>` ou o equivalente em C++ `<ciso646>` é preterido. No Microsoft C++, a `/permissive-` opção ou do `/Za` compilador é necessária para habilitar a grafia alternativa.

Exemplo

```
// expre_Logical_NOT_Operator.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

int main() {
    int i = 0;
    if (!i)
        cout << "i is zero" << endl;
}
```

Confira também

[Expressões com operadores unários](#)

[Operadores, precedência e Associação internos do C++](#)

[Operadores aritméticos unários](#)

Operador OR lógico: ||

02/09/2020 • 3 minutes to read • [Edit Online](#)

Sintaxe

`expressão || OR lógica expressão and lógica`

Comentários

O operador OR lógico (`||`) retornará o valor booleano `true` se um ou ambos os operandos for `true` e retornar de `false` outra forma. Os operandos são convertidos implicitamente em tipo `bool` antes da avaliação e o resultado é do tipo `bool`. O OR lógico tem associatividade da esquerda para a direita.

Os operandos para o operador OR lógico não precisam ter o mesmo tipo, mas devem ser do tipo booleano, integral ou de ponteiro. Os operandos são geralmente expressões relacionais ou de igualdade.

O primeiro operando é completamente avaliado e todos os efeitos colaterais são concluídos antes de continuar a avaliação da expressão OR lógica.

O segundo operando será avaliado somente se o primeiro operando for avaliado como `false`, porque a avaliação não é necessária quando a expressão or lógica é `true`. Ele é conhecido como avaliação de *curto-circuito*.

```
printf( "%d" , (x == w || x == y || x == z) );
```

No exemplo acima, se `x` é igual a ou, `w`, `y` ou `z`, o segundo argumento para a função é `printf` avaliado como `true`, que é promovido para um número inteiro e o valor 1 é impresso. Caso contrário, ele é avaliado como `false` e o valor 0 é impresso. Assim que uma das condições for avaliada como `true`, a avaliação será interrompida.

Palavra-chave Operator para ||

O C++ especifica `or` como uma grafia alternativa para `||`. Em C, a grafia alternativa é fornecida como uma macro no `<iso646.h>` cabeçalho. Em C++, a grafia alternativa é uma palavra-chave; o uso do `<iso646.h>` ou o equivalente em C++ `<ciso646>` é preterido. No Microsoft C++, a `/permissive-` opção ou do `/Za` compilador é necessária para habilitar a grafia alternativa.

Exemplo

```
// expre_Logical_OR_Operator.cpp
// compile with: /EHsc
// Demonstrate logical OR
#include <iostream>
using namespace std;
int main() {
    int a = 5, b = 10, c = 15;
    cout << boolalpha
        << "The true expression "
        << "a < b || b > c yields "
        << (a < b || b > c) << endl
        << "The false expression "
        << "a > b || b > c yields "
        << (a > b || b > c) << endl;
}
```

Confira também

[Operadores, precedência e Associação internos do C++](#)

[Operadores lógicos C](#)

Operadores de acesso de membro: . e ->

25/03/2020 • 2 minutes to read • [Edit Online](#)

Sintaxe

```
postfix-expression . name
postfix-expression -> name
```

Comentários

Os operadores de acesso de membro . e -> são usados para fazer referência a membros de estruturas, uniões e classes. As expressões de acesso do membro têm o valor e o tipo do membro selecionado.

Há duas formas de expressões de acesso do membro:

1. No primeiro formulário, o *sufixo-expressão* representa um valor de struct, classe ou tipo de União, e *nome* nomeia um membro da estrutura, União ou classe especificada. O valor da operação é o de *Name* e é um I-value se o *sufixo-Expression* é um I-Value.
2. No segundo formulário, o *sufixo-Expression* representa um ponteiro para uma estrutura, União ou classe e *nome* nomeia um membro da estrutura, União ou classe especificada. O valor é o de *Name* e é um I-Value. O operador -> faz a referência ao ponteiro. Portanto, as expressões `e->member` e `(*e).member` (onde *e* representa um ponteiro) produzem resultados idênticos (exceto quando os operadores -> ou * estão sobrecarregados).

Exemplo

O exemplo a seguir demonstra as duas formas de operador de acesso do membro.

```
// expre_Selection_Operator.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

struct Date {
    Date(int i, int j, int k) : day(i), month(j), year(k){}
    int month;
    int day;
    int year;
};

int main() {
    Date mydate(1,1,1900);
    mydate.month = 2;
    cout << mydate.month << "/" << mydate.day
        << "/" << mydate.year << endl;

    Date *mydate2 = new Date(1,1,2000);
    mydate2->month = 2;
    cout << mydate2->month << "/" << mydate2->day
        << "/" << mydate2->year << endl;
    delete mydate2;
}
```

2/1/1900

2/1/2000

Confira também

[Expressões pós-fixadas](#)

[Operadores internos, precedência e associatividade C++](#)

[Classes e Structs](#)

[Membros de união e estrutura](#)

Operadores multiplicativos e o operador de módulo

02/09/2020 • 4 minutes to read • [Edit Online](#)

Sintaxe

```
expression * expression
expression / expression
expression % expression
```

Comentários

Os operadores de multiplicação são:

- Multiplicação (*)
- Divisão (/)
- Módulo (restante da divisão) (%)

Esses operadores binários possuem associatividade da esquerda para a direita.

Os operadores de multiplicação usam operandos de tipos aritméticos. O operador de módulo (%) tem um requisito mais estrito no que seus operandos devem ser do tipo integral. (Para obter o restante de uma divisão de ponto flutuante, use a função de tempo de execução, [fmod](#).) As conversões abordadas nas [conversões padrão](#) são aplicadas aos operandos e o resultado é do tipo convertido.

O operador de multiplicação gera o resultado da multiplicação do primeiro operando pelo segundo.

O operador de divisão gera o resultado da divisão do primeiro operando pelo segundo.

O operador de módulo produz o restante fornecido pela expressão a seguir, em que $E1$ é o primeiro operando e $E2$ é o segundo: $E1 - (E1 / E2) * E2$, em que ambos os operandos são de tipos inteiros.

A divisão por 0 em uma divisão ou em uma expressão de módulo é indefinida e gera um erro de tempo de execução. Desse modo, as expressões a seguir geram resultados indefinidos e errôneos:

```
i % 0
f / 0.0
```

Se ambos os operandos em uma expressão de multiplicação, divisão ou módulo tiverem o mesmo sinal, o resultado será positivo. Caso contrário, o resultado será negativo. O resultado do sinal de uma operação de módulo é definido pela implementação.

NOTE

Uma vez que as conversões executadas pelos operadores de multiplicação não fornecem condições de estouro ou estouro negativo, as informações poderão ser perdidas se o resultado de uma operação de multiplicação não puder ser representado no tipo dos operandos após a conversão.

Específico da Microsoft

No Microsoft C++, o resultado de uma expressão de módulo sempre é igual ao sinal do primeiro operando.

FINAL específico da Microsoft

Se a divisão calculada de dois inteiros for inexata e apenas um operando for negativo, o resultado será o interior maior (em magnitude, independentemente do sinal), que é menor que o valor exato que seria gerado pela operação de divisão. Por exemplo, o valor calculado de $-11/3$ é $-3,666666666$. O resultado dessa divisão integral é 3.

A relação entre os operadores multiplicativa é fornecida pela identidade $(E1 / E2) * E2 + E1 \% E2 == E1$.

Exemplo

O programa a seguir demonstra os operadores de multiplicação. Observe que qualquer operando de `10 / 3` deve ser convertido explicitamente para o tipo `float` para evitar truncamento para que ambos os operandos sejam do tipo `float` antes da divisão.

```
// expr_Multiplicative_Operators.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;
int main() {
    int x = 3, y = 6, z = 10;
    cout << "3 * 6 is " << x * y << endl
        << "6 / 3 is " << y / x << endl
        << "10 % 3 is " << z % x << endl
        << "10 / 3 is " << (float) z / x << endl;
}
```

Confira também

[Expressões com operadores binários](#)

[Operadores internos C++, precedência e associatividade](#)

[Operadores de multiplicativa de C](#)

Operador new (C++)

02/09/2020 • 14 minutes to read • [Edit Online](#)

Aloca memória para um objeto ou matriz de objetos do *tipo nome* da loja gratuita e retorna um ponteiro digitado adequadamente, diferente de zero para o objeto.

NOTE

As extensões de componente do Microsoft C++ fornecem suporte para a `new` palavra-chave para adicionar entradas de slot vtable. Para obter mais informações, consulte [novo \(novo slot em vtable\)](#)

Sintaxe

```
[::] new [placement] new-type-name [new-initializer]  
[::] new [placement] ( type-name ) [new-initializer]
```

Comentários

Se não for bem-sucedida, `new` retorna zero ou gera uma exceção; consulte [os operadores novo e excluir](#) para obter mais informações. Você pode alterar esse comportamento padrão escrevendo uma rotina de tratamento de exceção personalizada e chamando a função de biblioteca de tempo de execução `_set_new_handler` com seu nome de função como seu argumento.

Para obter informações sobre como criar um objeto no heap gerenciado, consulte [gcnew](#).

Quando `new` é usado para alocar memória para um objeto de classe C++, o construtor do objeto é chamado depois que a memória é alocada.

Use o operador `delete` para desalocar a memória alocada com o `new` operador.

O exemplo a seguir aloca e depois libera uma matriz bidimensional de caracteres de tamanho `dim` por 10. Ao alocar uma matriz multidimensional, todas as dimensões, exceto a primeira, devem ser expressões de constantes que são avaliadas como valores positivos; a dimensão mais à esquerda da matriz pode ser qualquer expressão que seja avaliada como um valor positivo. Ao alocar uma matriz usando o `new` operador, a primeira dimensão pode ser zero — o `new` operador retorna um ponteiro exclusivo.

```
char (*pchar)[10] = new char[dim][10];  
delete [] pchar;
```

O *nome do tipo* não pode conter `const`, `volatile`, declarações de classe ou declarações de enumeração. Dessa forma, a expressão a seguir é inválida:

```
volatile char *vch = new volatile char[20];
```

O `new` operador não aloca tipos de referência porque eles não são objetos.

O `new` operador não pode ser usado para alocar uma função, mas pode ser usado para alocar ponteiros para funções. O exemplo a seguir aloca e depois libera uma matriz de sete ponteiros para funções que retornam inteiros.

```
int (**p) () = new (int (*[7]) ());
delete *p;
```

Se você usar o operador `new` sem nenhum argumento extra e compilar com a opção `/GX`, `/EHA` ou `o/EHS`, o compilador irá gerar o código para chamar o operador `delete` se o Construtor lançar uma exceção.

A lista a seguir descreve os elementos gramaticais de `new` :

Placement

Fornece uma maneira de passar argumentos adicionais se você sobrecarregar `new`.

nome do tipo

Especifica o tipo a ser alocado; pode ser um tipo definido pelo usuário ou interno. Se a especificação de tipo for complicada, poderá ficar entre parênteses para forçar a ordem de associação.

initializer

Fornece um valor para o objeto inicializado. Não é possível especificar inicializadores para matrizes. O operador criará matrizes de objetos somente se a classe tiver um construtor padrão.

Exemplo

O exemplo de código a seguir aloca uma matriz de caracteres e um objeto da classe `CName` e depois os libera.

```

// expre_new_Operator.cpp
// compile with: /EHsc
#include <string.h>

class CName {
public:
    enum {
        sizeOfBuffer = 256
    };

    char m_szFirst[sizeOfBuffer];
    char m_szLast[sizeOfBuffer];

public:
    void SetName(char* pszFirst, char* pszLast) {
        strcpy_s(m_szFirst, sizeOfBuffer, pszFirst);
        strcpy_s(m_szLast, sizeOfBuffer, pszLast);
    }
};

int main() {
    // Allocate memory for the array
    char* pCharArray = new char[CName::sizeOfBuffer];
    strcpy_s(pCharArray, CName::sizeOfBuffer, "Array of characters");

    // Deallocate memory for the array
    delete [] pCharArray;
    pCharArray = NULL;

    // Allocate memory for the object
    CName* pName = new CName;
    pName->SetName("Firstname", "Lastname");

    // Deallocate memory for the object
    delete pName;
    pName = NULL;
}

```

Exemplo

Se você usar a nova forma de posicionamento do `new` operador, o formulário com argumentos além do tamanho da alocação, o compilador não oferecerá suporte a uma forma de posicionamento do `delete` operador se o Construtor lançar uma exceção. Por exemplo:

```

// expre_new_Operator2.cpp
// C2660 expected
class A {
public:
    A(int) { throw "Fail!"; }
};

void F(void) {
    try {
        // heap memory pointed to by pa1 will be deallocated
        // by calling ::operator delete(void*).
        A* pa1 = new A(10);
    } catch (...) {
    }
    try {
        // This will call ::operator new(size_t, char*, int).
        // When A::A(int) does a throw, we should call
        // ::operator delete(void*, char*, int) to deallocate
        // the memory pointed to by pa2. Since
        // ::operator delete(void*, char*, int) has not been implemented,
        // memory will be leaked when the deallocation cannot occur.

        A* pa2 = new(__FILE__, __LINE__) A(20);
    } catch (...) {
    }
}

int main() {
    A a;
}

```

Inicializando objeto alocado com novo

Um campo de *inicializador* opcional é incluído na gramática para o `new` operador. Isso permite que os novos objetos sejam inicializados com construtores definidos pelo usuário. Para obter mais informações sobre como a inicialização é feita, consulte [inicializadores](#). O exemplo a seguir ilustra como usar uma expressão de inicialização com o `new` operador:

```

// expre_Initializing_Objects_Allocated_with_new.cpp
class Acct
{
public:
    // Define default constructor and a constructor that accepts
    // an initial balance.
    Acct() { balance = 0.0; }
    Acct( double init_balance ) { balance = init_balance; }

private:
    double balance;
};

int main()
{
    Acct *CheckingAcct = new Acct;
    Acct *SavingsAcct = new Acct( 34.98 );
    double *HowMuch = new double( 43.0 );
    // ...
}

```

Neste exemplo, o objeto `CheckingAcct` é alocado usando o `new` operador, mas nenhuma inicialização padrão é especificada. Portanto, o construtor padrão da classe, `Acct()`, é chamado. O objeto `SavingsAcct` é então alocado da mesma maneira, mas ele é inicializado explicitamente como 34,98. Como 34,98 é do tipo `double`, o construtor que usa um argumento desse tipo é chamado para manipular a inicialização. Por fim, o tipo sem

classe `HowMuch` é inicializado como 43,0.

Se um objeto for de um tipo de classe e essa classe tiver construtores (como no exemplo anterior), o objeto poderá ser inicializado pelo `new` operador somente se uma dessas condições for atendida:

- Os argumentos fornecidos no inicializador concordam com os de um construtor.
- A classe tem um construtor padrão (um construtor um que pode ser chamado sem argumentos).

Nenhuma inicialização explícita por elemento pode ser feita ao alocar matrizes usando o `new` operador; somente o construtor padrão, se presente, é chamado. Consulte [argumentos padrão](#) para obter mais informações.

Se a alocação de memória falhar (`operator New` retorna um valor de 0), nenhuma inicialização é executada. Isso protege contra tentativas de inicialização de dados que não existem.

Assim como acontece com as chamadas de função, a ordem em que as expressões inicializadas são avaliadas não está definida. Além disso, você não deve confiar que essas expressões serão totalmente avaliadas antes que a alocação da memória seja executada. Se a alocação de memória falhar e o `new` operador retornar zero, algumas expressões no inicializador poderão não ser completamente avaliadas.

Tempo de vida dos objetos alocados com o novo

Os objetos alocados com o `new` operador não são destruídos quando o escopo no qual eles são definidos é encerrado. Como o `new` operador retorna um ponteiro para os objetos alocados, o programa deve definir um ponteiro com escopo adequado para acessar esses objetos. Por exemplo:

```
// expre_Lifetime_of_Objects_Allocated_with_new.cpp
// C2541 expected
int main()
{
    // Use new operator to allocate an array of 20 characters.
    char *AnArray = new char[20];

    for( int i = 0; i < 20; ++i )
    {
        // On the first iteration of the loop, allocate
        // another array of 20 characters.
        if( i == 0 )
        {
            char *AnotherArray = new char[20];
        }

        delete [] AnotherArray; // Error: pointer out of scope.
        delete [] AnArray;     // OK: pointer still in scope.
    }
}
```

Uma vez que o ponteiro `AnotherArray` sai do escopo no exemplo, o objeto não pode mais ser excluído.

Como funciona o novo

A expressão de *alocação* — a expressão que contém o `new` operador — faz três coisas:

- Localiza e reserva o armazenamento para que o objeto ou objetos sejam alocados. Quando essa fase for concluída, a quantidade correta de armazenamento é alocada, mas ela ainda não é um objeto.
- Inicializa o(s) objeto(s). Após a conclusão da inicialização, informações suficientes estão presentes para que o armazenamento alocado seja um objeto.

- Retorna um ponteiro para os objetos de um tipo de ponteiro derivado de *New-Type-Name* ou *Type-Name*. O programa usa esse ponteiro para acessar o objeto recentemente alocado.

O `new` operador invoca o operador de função **novo**. Para matrizes de qualquer tipo e para objetos que não são de `class`, `struct` ou `union` tipos, uma função global, `:: operator new`, é chamada para alocar armazenamento. Objetos de tipo de classe podem definir seu próprio **operador nova** função de membro estático em uma base por classe.

Quando o compilador encontra o `new` operador para alocar um objeto do tipo **Type**, ele emite uma chamada para `type :: operator new (sizeof (type))` ou, se nenhum **operador** definido pelo usuário novo estiver definido, `:: operator new (sizeof (type))`. Portanto, o `new` operador pode alocar a quantidade correta de memória para o objeto.

NOTE

O argumento para o **operador New** é do tipo `size_t`. Esse tipo é definido em, `<direct.h>` `<malloc.h>` `<memory.h>` `<search.h>` `<stddef.h>` `<stdio.h>` `<stdlib.h>` `<string.h>` e `<time.h>`.

Uma opção na gramática permite a especificação do *posicionamento* (consulte a gramática para **novo operador**). O parâmetro de *posicionamento* só pode ser usado para implementações definidas pelo usuário do **operador New**; Ele permite que informações extras sejam passadas para o **operador novo**. Uma expressão com um campo de *posicionamento* como `T *TObject = new (0x0040) T;` é convertida em

`T *TObject = T::operator new(sizeof(T), 0x0040);` se a classe T tem o operador membro novo, caso contrário `T *TObject = ::operator new(sizeof(T), 0x0040);`.

A intenção original do campo de *posicionamento* era permitir que os objetos dependentes de hardware sejam alocados em endereços especificados pelo usuário.

NOTE

Embora o exemplo anterior mostre apenas um argumento no campo de *posicionamento*, não há nenhuma restrição de quantos argumentos extras podem ser passados para o **operador novo** dessa forma.

Mesmo quando o **operador New** tiver sido definido para um tipo de classe, o operador global poderá ser usado usando a forma deste exemplo:

```
T *TObject = ::new TObject;
```

O operador de resolução de escopo (`::`) força o uso do `new` operador global.

Confira também

[Expressões com operadores unários](#)

[Palavras-chave](#)

[Operadores new e delete](#)

Operador de complemento individual: ~

02/09/2020 • 2 minutes to read • [Edit Online](#)

Sintaxe

```
~ cast-expression
```

Comentários

O operador de complemento de um (`~`), às vezes chamado de operador de *complemento bit-a-bit*, produz um complemento de um bit que de seu operando. Ou seja, cada bit que é 1 no operando, é 0 no resultado. De forma análoga, cada bit que é 0 no operando, é 1 no resultado. O operando para o operador de complemento de um deve ser um tipo integral.

Palavra-chave Operator para ~

O C++ especifica `compl` como uma grafia alternativa para `~`. Em C, a grafia alternativa é fornecida como uma macro no `<iso646.h>` cabeçalho. Em C++, a grafia alternativa é uma palavra-chave; o uso do `<iso646.h>` ou o equivalente em C++ `<ciso646>` é preterido. No Microsoft C++, a `/permissive-` opção ou do `/za` compilador é necessária para habilitar a grafia alternativa.

Exemplo

```
// expre_One_Complement_Operator.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;

int main () {
    unsigned short y = 0xFFFF;
    cout << hex << y << endl;
    y = ~y;    // Take one's complement
    cout << hex << y << endl;
}
```

Nesse exemplo, o novo valor atribuído a `y` é o complemento de um do valor sem sinal 0xFFFF, ou 0x0000.

A promoção integral é executada em operandos integrais. O tipo do operando é promovido para é o tipo resultante. Para obter mais informações sobre a promoção integral, consulte [conversões padrão](#).

Confira também

[Expressões com operadores unários](#)

[Operadores, precedência e Associação internos do C++](#)

[Operadores aritméticos unários](#)

Operadores de ponteiro para membro: `.*` e `->*`

15/04/2020 • 4 minutes to read • [Edit Online](#)

Sintaxe

```
expression .* expression
expression ->* expression
```

Comentários

Os operadores de ponteiro para membro, `.*` e `->`, retornam o valor de um membro de classe específico para o objeto especificado no lado esquerdo da expressão. O lado direito deve especificar um membro da classe. O exemplo a seguir mostra como usar estes operadores:

```
// expre_Expressions_with_Pointer_Member_Operators.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;

class Testpm {
public:
    void m_func1() { cout << "m_func1\n"; }
    int m_num;
};

// Define derived types pmfn and pmd.
// These types are pointers to members m_func1() and
// m_num, respectively.
void (Testpm::*pmfn)() = &Testpm::m_func1;
int Testpm::*pmd = &Testpm::m_num;

int main() {
    Testpm ATestpm;
    Testpm *pTestpm = new Testpm;

    // Access the member function
    (ATestpm.*pmfn)();
    (pTestpm->*pmfn)(); // Parentheses required since * binds
                          // less tightly than the function call.

    // Access the member data
    ATestpm.*pmd = 1;
    pTestpm->*pmd = 2;

    cout << ATestpm.*pmd << endl
        << pTestpm->*pmd << endl;
    delete pTestpm;
}
```

Saída

```
m_func1
m_func1
1
2
```

No exemplo anterior, um ponteiro para um membro, `pmfn`, é usado para invocar a função de membro `m_func1`.

Outro ponteiro para um membro, `pmd`, é usado para acessar o membro `m_num`.

O operador binário `.*` combina seu primeiro operando, que deve ser um objeto de tipo de classe, com o segundo operando, que deve ser um tipo de ponteiro para membro.

O operador binário `->*` combina seu primeiro operando, que deve ser um ponteiro para um objeto de tipo de classe, com seu segundo operando, que deve ser um tipo de ponteiro-para-membro.

Em uma expressão que contém o operador `.*`, o primeiro operando deve ser do tipo de classe, e é acessível para, do ponteiro para o membro especificado no segundo operando ou de um tipo acessível derivado de maneira não ambígua de e acessíveis a essa classe.

Em uma expressão contendo o operador `->*`, o primeiro operando deve ser do tipo "ponteiro para o tipo de classe" do tipo especificado no segundo operante, ou deve ser de um tipo inequivocamente derivado dessa classe.

Exemplo

Considere as seguintes classes e fragmento de programa:

```
// expe_Expressions_with_Pointer_Member_Operators2.cpp
// C2440 expected
class BaseClass {
public:
    BaseClass(); // Base class constructor.
    void Func1();
};

// Declare a pointer to member function Func1.
void (BaseClass::*pmfnFunc1)() = &BaseClass::Func1;

class Derived : public BaseClass {
public:
    Derived(); // Derived class constructor.
    void Func2();
};

// Declare a pointer to member function Func2.
void (Derived::*pmfnFunc2)() = &Derived::Func2;

int main() {
    BaseClass ABase;
    Derived ADerived;

    (ABase.*pmfnFunc1)(); // OK: defined for BaseClass.
    (ABase.*pmfnFunc2)(); // Error: cannot use base class to
                          // access pointers to members of
                          // derived classes.

    (ADerived.*pmfnFunc1)(); // OK: Derived is unambiguously
                            // derived from BaseClass.
    (ADerived.*pmfnFunc2)(); // OK: defined for Derived.
}
```

O resultado dos operadores `*` de ponteiro para membro do `.*` ou `->*` é um objeto ou função do tipo especificado na declaração do ponteiro para membro. Assim, no exemplo anterior, o resultado da expressão

`ADerived.*pmfnFunc1()` é um ponteiro para uma função que retorna void. Esse resultado será um valor l se o segundo operando for um valor l.

NOTE

Se o resultado de um dos operadores de ponteiro para membro for uma função, o resultado poderá ser usado apenas como um operando para o operador da chamada de função.

Confira também

[Operadores internos C++, precedência e associatividade](#)

Operadores de incremento e de decremento pós-fixados: ++ e --

02/09/2020 • 3 minutes to read • [Edit Online](#)

Sintaxe

```
postfix-expression ++
postfix-expression --
```

Comentários

O C++ fornece operadores de incremento e decremento de prefixo e sufixo; esta seção descreve somente os operadores de incremento e decremento de sufixo. (Para obter mais informações, consulte [incremento de prefixo e diminuir operadores](#).) A diferença entre os dois é que, na notação de sufixo, o operador aparece após o *sufixo-expressão*, enquanto na notação de prefixo, o operador aparece antes da *expressão*. O exemplo a seguir mostra um operador de incremento de sufixo:

```
i++;
```

O efeito de aplicar o operador de incremento de sufixo (`++`) é que o valor do operando é aumentado por uma unidade do tipo apropriado. Da mesma forma, o efeito da aplicação do operador decremento de sufixo (`--`) é que o valor do operando é diminuído por uma unidade do tipo apropriado.

É importante observar que uma expressão de incremento ou decréscimo de sufixo é avaliada como o valor da expressão *antes* do aplicativo do respectivo operador. A operação de aumento ou diminuição ocorre *depois* que o operando é avaliado. Esse problema surge apenas quando a operação de incremento ou decremento de sufixo ocorre no contexto de uma expressão maior.

Quando um operador de sufixo é aplicado a um argumento de função, o valor do argumento não tem garantia de incremento ou decremento antes de ser passada para a função. Consulte a seção 1.9.17 no padrão C++ para obter mais informações.

Aplicar o operador de incremento de sufixo a um ponteiro para uma matriz de objetos do tipo `long` realmente adiciona quatro à representação interna do ponteiro. Esse comportamento faz com que o ponteiro, que anteriormente fazia referência ao elemento n th da matriz, refira-se ao elemento $(n+1)$ th.

Os operandos dos operadores sufixo e aumento de sufixo devem ser modificadores (não `const`) l-valores de tipo aritmético ou de ponteiro. O tipo do resultado é o mesmo da *expressão de sufixo*, mas não é mais um l-Value.

Visual Studio 2017 versão 15,3 e posterior (disponível com [/std: c++ 17](#)): o operando de um incremento de sufixo ou um operador de decréscimo não pode ser do tipo `bool`.

O código a seguir ilustra o operador de incremento de sufixo:

```
// expre_Postfix_Increment_and_Decrement_Operators.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

int main() {
    int i = 10;
    cout << i++ << endl;
    cout << i << endl;
}
```

As operações postincrement e postdecrement em tipos enumerados não têm suporte:

```
enum Compass { North, South, East, West };
Compass myCompass;
for( myCompass = North; myCompass != West; myCompass++ ) // Error
```

Confira também

[Expressões de sufixo](#)

[Operadores internos C++, precedência e associatividade](#)

[Operadores de aumento e diminuição de sufixo C](#)

Operadores de incremento e de decremento pré-fixados: `++` e `--`

02/09/2020 • 3 minutes to read • [Edit Online](#)

Sintaxe

```
++ unary-expression
-- unary-expression
```

Comentários

O operador de incremento de prefixo (`++`) adiciona um ao seu operando; esse valor incrementado é o resultado da expressão. O operando deve ser um l-Value que não seja do tipo `const`. O resultado é um valor l do mesmo tipo do operando.

O prefixo decrementar (`--`) é análogo ao operador de incremento de prefixo, exceto que o operando é decrementado por um e o resultado é esse valor diminuído.

Visual Studio 2017 versão 15,3 e posterior (disponível com [/std: c++ 17](#)): o operando de um incremento ou um operador de decréscimo não pode ser do tipo `bool` .

Os operadores de incremento e decremento de prefixo e sufixo afetam seus operandos. A principal diferença entre eles é a ordem em que o incremento ou decremento ocorre na avaliação de uma expressão. (Para obter mais informações, consulte [incremento de sufixo e diminuir operadores](#).) No formulário de prefixo, o incremento ou o decréscimo ocorre antes de o valor ser usado na avaliação da expressão, portanto, o valor da expressão é diferente do valor do operando. Na forma de sufixo, o incremento ou decremento ocorre após que o valor é usado na avaliação da expressão, assim o valor da expressão é igual ao valor do operando. Por exemplo, o programa seguir imprime " `++i = 6` ":

```
// expre_Increment_and_Decrement_Operators.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;

int main() {
    int i = 5;
    cout << "++i = " << ++i << endl;
}
```

Um operando tipo integral ou flutuante é incrementado ou decrementado pelo valor inteiro 1. O tipo do resultado é igual ao tipo do operando. Um operando do tipo ponteiro é incrementado ou decrementado pelo tamanho do objeto pertinente. Um ponteiro incrementado aponta para o próximo objeto; um ponteiro decrementado aponta para o objeto anterior.

Como os operadores de incremento e decremento têm efeitos colaterais, o uso de expressões com operadores de incremento ou decréscimo em uma [macro de pré-processador](#) pode ter resultados indesejáveis. Considere este exemplo:

```
// expre_Increment_and_Decrement_Operators2.cpp
#define max(a,b) ((a)<(b))?(b):(a)

int main()
{
    int i = 0, j = 0, k;
    k = max( ++i, j );
}
```

A macro se expande para:

```
k = ((++i)<(j))?(j):(++i);
```

Se `i` for maior ou igual a `j` ou menor que `j` por 1, será incrementado duas vezes.

NOTE

As funções embutidas C++ são preferíveis para macros em muitos casos, pois eliminam efeitos colaterais como os descritas aqui, e permitem que a linguagem faça uma verificação de tipo mais completa.

Confira também

[Expressões com operadores unários](#)

[Operadores internos C++, precedência e associatividade](#)

[Operadores de incremento e decréscimo de prefixo](#)

Operadores relacionais: < , > , < = e>=

02/09/2020 • 4 minutes to read • [Edit Online](#)

Sintaxe

```
expression < expression
expression > expression
expression <= expression
expression >= expression
```

Comentários

Os operadores relacionais binários determinam as seguintes relações:

- Menor que (<)
- Maior que (>)
- Menor que ou igual a (<=)
- Maior ou igual a (>=)

Esses operadores relacionais possuem associatividade da esquerda para a direita. Ambos os operandos de operadores relacionais devem ser do tipo aritmético ou de ponteiro. Eles geram valores do tipo `bool`. O valor retornado será `false` (0) se a relação na expressão for `false`; caso contrário, o valor retornado será `true` (1).

Exemplo

```
// expre_Relational_Operators.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;

int main() {
    cout << "The true expression 3 > 2 yields: "
        << (3 > 2) << endl
        << "The false expression 20 < 10 yields: "
        << (20 < 10) << endl;
}
```

As expressões no exemplo anterior devem ser colocadas entre parênteses porque o operador de inserção de fluxo (`<<`) tem precedência mais alta do que os operadores relacionais. Portanto, a primeira expressão sem parênteses seria avaliada como:

```
(cout << "The true expression 3 > 2 yields: " << 3) < (2 << "\n");
```

As conversões aritméticas usuais abordadas em [conversões padrão](#) são aplicadas a operandos de tipos aritméticos.

Comparando ponteiros

Quando dois ponteiros para objetos do mesmo tipo são comparados, o resultado é determinado pelo local dos objetos apontados no espaço de endereço do programa. Os ponteiros também podem ser comparados a uma expressão constante que é avaliada como 0 ou como um ponteiro do tipo `void *`. Se uma comparação de ponteiro for feita em um ponteiro do tipo `void *`, o outro ponteiro será convertido implicitamente no tipo `void *`. Então, a comparação será feita.

Dois ponteiros de tipos diferentes não podem ser comparados, a menos que:

- Um tipo é um tipo de classe derivado de outro tipo.
- Pelo menos um dos ponteiros é explicitamente convertido (convertido) no tipo `void *`. (O outro ponteiro é implicitamente convertido em tipo `void *` para a conversão.)

Dois ponteiros do mesmo tipo que apontam para o mesmo objeto são obrigatoriamente comparados como iguais. Se dois ponteiros para membros não estáticos de um objeto são comparados, as seguintes regras se aplicam:

- Se o tipo de classe não for um `union`, e se os dois membros não forem separados por um *especificador de acesso*, como `public`, `protected` ou `private`, o ponteiro para o membro declarado por último comparará maior do que o ponteiro para o membro declarado anteriormente.
- Se os dois membros forem separados por um *especificador de acesso*, os resultados serão indefinidos.
- Se o tipo de classe for a `union`, os ponteiros para membros de dados diferentes nesse `union` comparação serão iguais.

Se dois ponteiros apontarem para elementos da mesma matriz ou para o elemento além do final da matriz, o ponteiro para o objeto com o subscrito mais alto será comparado como superior. A comparação dos ponteiros é garantida como válida somente quando os ponteiros se referem a objetos na mesma matriz ou ao local após o término da matriz.

Confira também

[Expressões com operadores binários](#)

[Operadores internos C++, precedência e associatividade](#)

[Operadores de igualdade e relação C](#)

Operador de resolução do escopo: ::

25/03/2020 • 3 minutes to read • [Edit Online](#)

O operador de resolução de escopo :: é usado para identificar e desambiguar identificadores usados em escopos diferentes. Para obter mais informações sobre escopo, consulte [escopo](#).

Sintaxe

```
:: identifier
class-name :: identifier
namespace :: identifier
enum class :: identifier
enum struct :: identifier
```

Comentários

O `identifier` pode ser uma variável, uma função ou um valor de enumeração.

Com classes e namespaces

O exemplo a seguir mostra como o operador de resolução do escopo é usado com namespaces e classes:

```
namespace NamespaceA{
    int x;
    class ClassA {
        public:
            int x;
    };
}

int main() {

    // A namespace name used to disambiguate
    NamespaceA::x = 1;

    // A class name used to disambiguate
    NamespaceA::ClassA a1;
    a1.x = 2;
}
```

Um operador de resolução do escopo sem um qualificador de escopo refere-se ao namespace global.

```

namespace NamespaceA{
    int x;
}

int x;

int main() {
    int x;

    // the x in main()
    x = 0;
    // The x in the global namespace
    ::x = 1;

    // The x in the A namespace
    NamespaceA::x = 2;
}

```

Você pode usar o operador de resolução do escopo para identificar um membro de um namespace ou para identificar um namespace que nomeia namespace de membros em uma diretiva de uso. No exemplo abaixo, você pode usar `NamespaceC` para qualificar `ClassB`, embora `ClassB` tenha sido declarada em namespace `NamespaceB` porque `NamespaceB` foi nomeado em `NamespaceC` por uma diretiva de uso.

```

namespace NamespaceB {
    class ClassB {
        public:
            int x;
    };
}

namespace NamespaceC{
    using namespace B;
}
int main() {
    NamespaceB::ClassB c_b;
    NamespaceC::ClassB c_c;

    c_b.x = 3;
    c_c.x = 4;
}

```

Você pode usar cadeias de operadores de resolução do escopo. No exemplo a seguir, `NamespaceD::NamespaceD1` identifica o namespace aninhado `NamespaceD1` e `NamespaceE::ClassE::ClassE1` identifica a classe aninhada `ClassE1`.

```

namespace NamespaceD{
    namespace NamespaceD1{
        int x;
    }
}

namespace NamespaceE{
    class ClassE{
    public:
        class ClassE1{
        public:
            int x;
        };
    };
}

int main() {
    NamespaceD:: NamespaceD1::x = 6;
    NamespaceE::ClassE::ClassE1 e1;
    e1.x = 7 ;
}

```

Com membros estáticos

Você deve usar o operador de resolução do escopo para chamar membros estáticos de classes.

```

class ClassG {
public:
    static int get_x() { return x;}
    static int x;
};

int ClassG::x = 6;

int main() {

    int gx1 = ClassG::x;
    int gx2 = ClassG::get_x();
}

```

Com enumerações de escopo

O operador de resolução com escopo também é usado com os valores de [declarações de enumeração](#) de enumeração com escopo definido, como no exemplo a seguir:

```

enum class EnumA{
    First,
    Second,
    Third
};

int main() {
    EnumA enum_value = EnumA::First;
}

```

Confira também

[Operadores internos, precedência e associatividade C++](#)
[Namespaces](#)

Operador sizeof

02/09/2020 • 4 minutes to read • [Edit Online](#)

Produz o tamanho de seu operando em relação ao tamanho do tipo `char` .

NOTE

Para obter informações sobre o `sizeof ...` operador, consulte [reticências and Variadic templates](#).

Sintaxe

```
sizeof unary-expression
sizeof ( type-name )
```

Comentários

O resultado do `sizeof` operador é do tipo `size_t` , um tipo integral definido no arquivo de inclusão `<stddef.h>` . Esse operador permite que você evite especificar tamanhos de dados dependentes do computador em seus programas.

O operando `sizeof` pode ser um dos seguintes:

- O nome de um tipo. Para usar `sizeof` com um nome de tipo, o nome deve ser colocado entre parênteses.
- Uma expressão. Quando usado com uma expressão, `sizeof` pode ser especificado com ou sem os parênteses. A expressão não é avaliada.

Quando o `sizeof` operador é aplicado a um objeto do tipo `char` , ele resulta em 1. Quando o `sizeof` operador é aplicado a uma matriz, ele gera o número total de bytes nessa matriz, não o tamanho do ponteiro representado pelo identificador da matriz. Para obter o tamanho do ponteiro representado pelo identificador de matriz, passe-o como um parâmetro para uma função que usa `sizeof` . Por exemplo:

Exemplo

```

#include <iostream>
using namespace std;

size_t getPtrSize( char *ptr )
{
    return sizeof( ptr );
}

int main()
{
    char szHello[] = "Hello, world!";

    cout << "The size of a char is: "
        << sizeof( char )
        << "\nThe length of " << szHello << " is: "
        << sizeof szHello
        << "\nThe size of the pointer is "
        << getPtrSize( szHello ) << endl;
}

```

Saída de exemplo

```

The size of a char is: 1
The length of Hello, world! is: 14
The size of the pointer is 4

```

Quando o `sizeof` operador é aplicado a um `class` `struct` tipo, ou `union`, o resultado é o número de bytes em um objeto desse tipo, além de qualquer preenchimento adicionado para alinhar Membros em limites de palavras. O resultado não corresponde necessariamente ao tamanho calculado pela adição dos requisitos de armazenamento dos membros individuais. A opção do compilador `/ZP` e o pragma do `pacote` afetam os limites de alinhamento dos membros.

O `sizeof` operador nunca produz 0, mesmo para uma classe vazia.

O `sizeof` operador não pode ser usado com os seguintes operandos:

- Funções. (No entanto, `sizeof` pode ser aplicado a ponteiros para funções.)
- Campos de bits.
- Classes indefinidas.
- O tipo `void`.
- Matrizes alocadas dinamicamente.
- Matrizes externas.
- Tipos incompletos.
- Nomes entre parênteses de tipos incompletos.

Quando o `sizeof` operador é aplicado a uma referência, o resultado é o mesmo que se `sizeof` tivesse sido aplicado ao próprio objeto.

Se uma matriz sem tamanho for o último elemento de uma estrutura, o `sizeof` operador retornará o tamanho da estrutura sem a matriz.

O `sizeof` operador geralmente é usado para calcular o número de elementos em uma matriz usando uma expressão do formulário:

```
sizeof array / sizeof array[0]
```

Confira também

[Expressões com operadores unários](#)

[Palavras-chave](#)

Operador subscrito []

02/09/2020 • 6 minutes to read • [Edit Online](#)

Sintaxe

```
postfix-expression [ expression ]
```

Comentários

Uma expressão de sufixo (que também pode ser uma expressão primária) seguida pelo operador subscrito, [], especifica a indexação de matriz.

Para obter informações sobre matrizes gerenciadas em C++/CLI, consulte [matrizes](#).

Normalmente, o valor representado pelo *sufixo-Expression* é um valor de ponteiro, como um identificador de matriz, e *expression* é um valor integral (incluindo tipos enumerados). No entanto, tudo o que é necessário sintaticamente é que uma das expressões seja do tipo ponteiro e a outra seja do tipo integral. Portanto, o valor integral pode estar na posição da *expressão de sufixo* e o valor do ponteiro pode estar entre colchetes na posição da *expressão* ou do subscrito. Considere o fragmento de código a seguir:

```
int nArray[5] = { 0, 1, 2, 3, 4 };
cout << nArray[2] << endl;           // prints "2"
cout << 2[nArray] << endl;         // prints "2"
```

No exemplo acima, a expressão `nArray[2]` é idêntica a `2[nArray]`. O motivo é que o resultado de uma expressão de subscrito `e1[e2]` é fornecido por:

```
*((e2) + (e1))
```

O endereço produzido pela expressão não é *E2* bytes do endereço *E1*. Em vez disso, o endereço é dimensionado para produzir o próximo objeto na matriz *E2*. Por exemplo:

```
double aDb1[2];
```

Os endereços de `aDb[0]` e `aDb[1]` têm 8 bytes de distância — o tamanho de um objeto do tipo `double`. Esse dimensionamento de acordo com o tipo de objeto é feito automaticamente pela linguagem C++ e é definido em [operadores aditivos](#) onde a adição e a subtração de operandos do tipo de ponteiro são discutidas.

Uma expressão subscrita também pode ter vários subscritos, como segue:

```
expression1 [ expression2 ] [ expression3 ] ...
```

As expressões subscritas são associadas da esquerda para a direita. A expressão de subscrito mais à esquerda, *expression1 [expression2]*, é avaliada primeiro. O endereço resultante da adição de *expression1* e *expression2* forma uma expressão do ponteiro; *expression3* é adicionada a essa expressão de ponteiro para formar uma nova expressão de ponteiro e assim por diante até que a última expressão subscrita seja adicionada. O operador de indireção (`*`) é aplicado depois que a última expressão de subscrito é avaliada, a menos que o valor final do ponteiro atenda a um tipo de matriz.

As expressões com vários subscritos referem-se aos elementos de matrizes multidimensionais. Uma matriz

multidimensional é uma matriz cujos elementos são matrizes. Por exemplo, o primeiro elemento de uma matriz tridimensional é uma matriz com duas dimensões. O exemplo a seguir declara e inicializa uma matriz bidimensional simples de caracteres:

```
// expre_Subscript_Operator.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
#define MAX_ROWS 2
#define MAX_COLS 2

int main() {
    char c[ MAX_ROWS ][ MAX_COLS ] = { { 'a', 'b' }, { 'c', 'd' } };
    for ( int i = 0; i < MAX_ROWS; i++ )
        for ( int j = 0; j < MAX_COLS; j++ )
            cout << c[ i ][ j ] << endl;
}
```

Subscritos positivos e negativos

O primeiro elemento de uma matriz é o elemento 0. O intervalo de uma matriz de C++ é da `matriz[0]` para a `matriz[size - 1]`. No entanto, o C++ oferece suporte a subscritos positivos e negativos. Os subscritos negativos devem estar dentro dos limites da matriz, se não estiverem, os resultados serão imprevisíveis. O código a seguir mostra os subscritos positivo e negativo da matriz:

```
#include <iostream>
using namespace std;

int main() {
    int intArray[1024];
    for ( int i = 0, j = 0; i < 1024; i++ )
    {
        intArray[i] = j++;
    }

    cout << intArray[512] << endl;    // 512
    cout << 257[intArray] << endl;    // 257

    int *midArray = &intArray[512]; // pointer to the middle of the array

    cout << midArray[-256] << endl; // 256
    cout << intArray[-256] << endl; // unpredictable, may crash
}
```

O subscrito negativo na última linha pode produzir um erro em tempo de execução porque aponta para um endereço 256 `int` posições inferiores na memória do que a origem da matriz. O ponteiro `midArray` é inicializado com o meio de `intArray`; portanto, é possível (mas perigoso) usar índices de matriz positivos e negativos nele. Os erros de subscrito de matriz não geram erros de tempo de compilação, mas geram resultados imprevisíveis.

O operador subscrito é comutativo. Portanto, é garantido que a `matriz[index]` e o `índice[array]` sejam equivalentes, desde que o operador subscrito não esteja sobrecarregado (consulte [operadores sobrecarregados](#)). O primeiro formulário é a prática de codificação mais comum, mas ambas funcionam.

Confira também

[Expressões de sufixo](#)

Operadores internos C++, precedência e associatividade

matrizes

Matrizes unidimensionais

Matrizes multidimensionais

Operador typeid

02/09/2020 • 4 minutes to read • [Edit Online](#)

Sintaxe

```
typeid(type-id)
typeid(expression)
```

Comentários

O `typeid` operador permite que o tipo de um objeto seja determinado em tempo de execução.

O resultado de `typeid` é um `const type_info&`. O valor é uma referência a um `type_info` objeto que representa o *tipo-ID* ou o tipo da *expressão*, dependendo de qual forma de `typeid` é usada. Para obter mais informações, consulte [classe type_info](#).

O `typeid` operador não funciona com tipos gerenciados (declaradores abstratos ou instâncias). Para obter informações sobre como obter o *Type* de um tipo especificado, consulte [typeid](#).

O `typeid` operador faz uma verificação de tempo de execução quando aplicado a um l-Value de um tipo de classe polimórfico, em que o tipo true do objeto não pode ser determinado pelas informações estáticas fornecidas. Tais casos são:

- Uma referência à classe
- Um ponteiro, desreferenciado com `*`
- Um ponteiro subscrito (`[]`). (Não é seguro usar um subscrito com um ponteiro para um tipo polimórfico.)

Se a *expressão* apontar para um tipo de classe base, mas o objeto for, na verdade, de um tipo derivado dessa classe base, uma `type_info` referência para a classe derivada será o resultado. A *expressão* deve apontar para um tipo polimórfico (uma classe com funções virtuais). Caso contrário, o resultado será o `type_info` para a classe estática referenciada na *expressão*. Além disso, o ponteiro deve ser desreferenciado para que o objeto usado seja aquele para o qual ele aponta. Sem desreferenciar o ponteiro, o resultado será o `type_info` para o ponteiro, não para o que ele aponta. Por exemplo:

```

// expre_typeid_Operator.cpp
// compile with: /GR /EHsc
#include <iostream>
#include <typeinfo>

class Base {
public:
    virtual void vfunc() {}
};

class Derived : public Base {};

using namespace std;
int main() {
    Derived* pd = new Derived;
    Base* pb = pd;
    cout << typeid( pb ).name() << endl;    //prints "class Base *"
    cout << typeid( *pb ).name() << endl;    //prints "class Derived"
    cout << typeid( pd ).name() << endl;    //prints "class Derived *"
    cout << typeid( *pd ).name() << endl;    //prints "class Derived"
    delete pd;
}

```

Se a *expressão* estiver desreferenciando um ponteiro e o valor do ponteiro for zero, `typeid` o lançará uma exceção `bad_typeid`. Se o ponteiro não apontar para um objeto válido, uma `__non_rtti_object` exceção será lançada. Indica uma tentativa de analisar o RTTI que disparou uma falha porque o objeto é de alguma forma inválido. (Por exemplo, é um ponteiro incorreto ou o código não foi compilado com `/gr`).

Se a *expressão* não for um ponteiro, e não uma referência a uma classe base do objeto, o resultado será uma `type_info` referência que representa o tipo estático da *expressão*. O *tipo estático* de uma expressão refere-se ao tipo de uma expressão como é conhecido no momento da compilação. A semântica da execução é ignorada ao avaliar o tipo estático de uma expressão. Além disso, as referências são ignoradas quando possível para determinar o tipo estático de uma expressão:

```

// expre_typeid_Operator_2.cpp
#include <typeinfo>

int main()
{
    typeid(int) == typeid(int&); // evaluates to true
}

```

`typeid` também pode ser usado em modelos para determinar o tipo de um parâmetro de modelo:

```

// expre_typeid_Operator_3.cpp
// compile with: /c
#include <typeinfo>
template < typename T >
T max( T arg1, T arg2 ) {
    cout << typeid( T ).name() << "s compared." << endl;
    return ( arg1 > arg2 ? arg1 : arg2 );
}

```

Confira também

[Informações de tipo de tempo de execução](#)

[Palavras-chave](#)

Operadores unários de adição e de negação: + e -

02/09/2020 • 2 minutes to read • [Edit Online](#)

Sintaxe

```
+ cast-expression
- cast-expression
```

Operador +

O resultado do operador unário (+) é o valor de seu operando. O operando para o operador de adição unário deve ser de um tipo aritmético.

A promoção integral é executada em operandos inteiros. O tipo resultante é o tipo para o qual o operando é promovido. Portanto, a expressão `+ch` , em que `ch` é do tipo `char` , resulta em tipo `int` ; o valor não é modificado. Consulte [conversões padrão](#) para obter mais informações sobre como a promoção é feita.

Operador -

O operador de negação unário (-) produz o negativo de seu operando. O operando para o operador de negação unário deve ser um tipo aritmético.

A promoção de integral é executada em operandos inteiros, e o tipo resultante é o tipo para o qual o operando é promovido. Consulte [conversões padrão](#) para obter mais informações sobre como a promoção é executada.

Específico da Microsoft

A negação unária de quantidades não assinadas é executada subtraindo o valor do operando de 2^n , onde n é o número de bits em um objeto de um determinado tipo sem assinatura.

FINAL específico da Microsoft

Confira também

[Expressões com operadores unários](#)

[Operadores internos C++, precedência e associatividade](#)

Expressões (C++)

25/03/2020 • 2 minutes to read • [Edit Online](#)

Esta seção descreve as expressões C++. As expressões são sequências de operadores e operandos usadas em uma ou mais das seguintes finalidades:

- Cálculo de um valor a partir dos operandos.
- Designação de objetos ou funções.
- Geração de "efeitos colaterais". (Efeitos colaterais são todas as ações diferentes da avaliação da expressão — por exemplo, a alteração do valor de um objeto.)

Em C++, os operadores podem ser sobre carregados e seus significados podem ser definidos pelo usuário. No entanto, a precedência e o número de operandos que eles utilizam não podem ser alterados. Esta seção descreve a sintaxe e a semântica dos operadores como eles são fornecidos com a linguagem, não quando são sobre carregados. Além dos [tipos de expressões](#) e [semânticas de expressões](#), os seguintes tópicos são abordados:

- [Expressões primárias](#)
- [Operador de resolução de escopo](#)
- [Expressões de sufixo](#)
- [Expressões com operadores unários](#)
- [Expressões com operadores binários](#)
- [Operador condicional](#)
- [Expressões constantes](#)
- [Operadores de conversão](#)
- [Informações de tipo de tempo de execução](#)

Tópicos sobre operadores em outras seções:

- [Operadores internos, precedência e associatividade C++](#)
- [Operadores sobre carregados](#)
- [typeid \(C++/CLI\)](#)

NOTE

Os operadores para tipos internos não podem ser sobre carregados; o comportamento deles é predefinido.

Confira também

[Referência da linguagem C++](#)

Tipos de expressões

25/03/2020 • 2 minutes to read • [Edit Online](#)

As expressões C++ são divididas em várias categorias:

- [Expressões primárias](#). São os blocos de construção dos quais todas as outras expressões são formadas.
- [Expressões de sufixo](#). São expressões primárias seguidas por um operador — por exemplo, o subscrito de matriz ou o operador de incremento de sufixo.
- [Expressões formadas com operadores unários](#). Os operadores unários atuam somente em um operando em uma expressão.
- [Expressões formadas com operadores binários](#). Os operadores binários atuam em dois operandos em uma expressão.
- [Expressões com o operador condicional](#). O operador condicional é um operador ternário — o único operador desse tipo na linguagem C++ — e usa três operandos.
- [Expressões constantes](#). As expressões constantes são totalmente formadas por dados constantes.
- [Expressões com conversões de tipo explícitas](#). As conversões de tipo explícito podem ser usadas em expressões.
- [Expressões com operadores de ponteiro para membro](#).
- [Conversão](#). Conversões de tipo seguro podem ser usadas em expressões.
- [Informações de tipo de tempo de execução](#). Determine o tipo de um objeto durante a execução do programa.

Confira também

[Expressões](#)

Expressões primárias

02/09/2020 • 3 minutes to read • [Edit Online](#)

As expressões primárias são os blocos de construção de expressões mais complexas. Elas são literais, nomes e nomes qualificados pelo operador de resolução de escopo (`::`). Uma expressão primária pode ter qualquer uma das seguintes formas:

```
literal
this
name
::name ( expression )
```

Um *literal* é uma expressão primária constante. Seu tipo depende da forma de sua especificação. Consulte [literais](#) para obter informações completas sobre como especificar literais.

A `this` palavra-chave é um ponteiro para um objeto de classe. Ela está disponível nas funções membro não estáticas e aponta para a instância da classe para a qual a função foi invocada. A `this` palavra-chave não pode ser usada fora do corpo de uma função de membro de classe.

O tipo do `this` ponteiro é `type ** * const**` (em que `type` é o nome da classe) dentro das funções que não modificam especificamente o `this` ponteiro. O exemplo a seguir mostra declarações de função de membro e os tipos de `this` :

```
// expre_Primary_Expressions.cpp
// compile with: /LD
class Example
{
public:
    void Func();           // * const this
    void Func() const;     // const * const this
    void Func() volatile; // volatile * const this
};
```

Consulte [esse ponteiro](#) para obter mais informações sobre como modificar o tipo do `this` ponteiro.

O operador de resolução de escopo (`::`) seguido por um nome constitui uma expressão primária. Tais nomes devem estar no escopo global, não em nomes de membro. O tipo dessa expressão é determinado pela declaração do nome. Será um l-value (isto é, pode aparecer à esquerda de uma expressão de operador de atribuição) se o nome declarativo for um l-value. O operador de resolução de escopo permite que um nome global seja referenciado, mesmo se esse nome estiver oculto no escopo atual. Consulte [escopo](#) para obter um exemplo de como usar o operador de resolução de escopo.

Uma expressão entre parênteses é uma expressão primária cujo tipo e valor são idênticos aos da expressão fora dos parênteses. Será um l-value se a expressão fora dos parênteses for um l-value.

Os exemplos de expressões primárias incluem:

```
100 // literal
'c' // literal
this // in a member function, a pointer to the class instance
::func // a global function
::operator + // a global operator function
::A::B // a global qualified name
( i + 1 ) // a parenthesized expression
```

Os exemplos a seguir são todos considerados *nomes*, portanto, expressões primárias, em várias formas:

```
MyClass // a identifier
MyClass::f // a qualified name
operator = // an operator function name
operator char* // a conversion operator function name
~MyClass // a destructor name
A::B // a qualified name
A<int> // a template id
```

Confira também

[Tipos de expressões](#)

Reticências e modelos Variadic

02/09/2020 • 6 minutes to read • [Edit Online](#)

Este artigo mostra como usar reticências (`...`) com modelos variadic de C++. As reticências tiveram muitos usos em C e C++. Eles incluem listas de argumentos variáveis para funções. A função `printf()` da Biblioteca em Runtime C é um dos exemplos mais conhecidos.

Um *modelo Variadic* é um modelo de classe ou função que dá suporte a um número arbitrário de argumentos. Esse mecanismo é útil principalmente para os desenvolvedores de biblioteca C++, pois você pode aplicá-lo a modelos de classe e modelos de função e, dessa forma, fornecer uma ampla variedade de funcionalidades e flexibilidade fortemente tipadas e não triviais.

Sintaxe

As reticências são usadas em duas maneiras por modelos variadic. À esquerda do nome do parâmetro, ele significa um pacote de *parâmetros* à direita do nome do parâmetro, ele expande os pacotes de parâmetros em nomes separados.

Aqui está um exemplo básico de sintaxe de definição de *classe de modelo Variadic*:

```
template<typename... Arguments> class classname;
```

Para pacotes e expansões do parâmetro, você pode acrescentar espaço em branco em torno de reticências, com base em sua preferência, conforme mostrado nestes exemplos:

```
template<typename ...Arguments> class classname;
```

Ou assim:

```
template<typename ... Arguments> class classname;
```

Observe que este artigo usa a Convenção que é mostrada no primeiro exemplo (as reticências são anexadas a `typename`).

Nos exemplos anteriores, *arguments* é um pacote de parâmetros. A classe `classname` pode aceitar um número variável de argumentos, como nestes exemplos:

```
template<typename... Arguments> class vtclass;

vtclass< > vtinstance1;
vtclass<int> vtinstance2;
vtclass<float, bool> vtinstance3;
vtclass<long, std::vector<int>, std::string> vtinstance4;
```

Ao usar uma definição de classe de modelo variadic, você também pode requisitar pelo menos um parâmetro:

```
template <typename First, typename... Rest> class classname;
```

Veja um exemplo básico de sintaxe de *função de modelo Variadic*:

```
template <typename... Arguments> returntype functionname(Arguments... args);
```

O pacote de parâmetros de *argumentos* é então expandido para uso, conforme mostrado na próxima seção, entendendo modelos Variadic.

Outras formas de sintaxe de função de modelo variadic são possíveis, incluindo, mas não limitado a, esses exemplos:

```
template <typename... Arguments> returntype functionname(Arguments&... args);
template <typename... Arguments> returntype functionname(Arguments&&... args);
template <typename... Arguments> returntype functionname(Arguments*... args);
```

Especificadores como `const` também são permitidos:

```
template <typename... Arguments> returntype functionname(const Arguments&... args);
```

Tal como as definições de classe de modelo variadic, você pode criar funções que exigem ao menos um parâmetro:

```
template <typename First, typename... Rest> returntype functionname(const First& first, const Rest&... args);
```

Os modelos Variadic usam o operador `sizeof...()` (não relacionado ao operador mais antigo `sizeof()`):

```
template<typename... Arguments>
void tfunc(const Arguments&... args)
{
    constexpr auto numargs{ sizeof...(Arguments) };

    X xobj[numargs]; // array of some previously defined type X

    helper_func(xobj, args...);
}
```

Mais sobre o posicionamento das reticências

Anteriormente, esse artigo descreveu a colocação de reticências que define pacotes e expansões de parâmetros como "à esquerda do nome do parâmetro, significa um pacote de parâmetros, e à direita do nome do parâmetro, expande os pacotes de parâmetros em nomes separados". Isso é tecnicamente verdade, mas pode ser uma tradução confusa do código. Considere:

- Em uma lista de parâmetros de template (`template <parameter-list>`), `typename...` apresenta um pacote de parâmetros de modelo.
- Em uma cláusula de declaração de parâmetro (`func(parameter-list)`), uma reticências de "nível superior" apresenta um pacote de parâmetros de função e o posicionamento de reticências é importante:

```
// v1 is NOT a function parameter pack:
template <typename... Types> void func1(std::vector<Types...> v1);

// v2 IS a function parameter pack:
template <typename... Types> void func2(std::vector<Types>... v2);
```

- Onde as reticências aparecem imediatamente após o nome de parâmetro, você tem uma expansão do pacote de parâmetros.

Exemplo

Uma boa maneira de ilustrar o mecanismo de função do modelo variadic é usá-lo em uma reescrita de algumas das funcionalidades de `printf`:

```
#include <iostream>

using namespace std;

void print() {
    cout << endl;
}

template <typename T> void print(const T& t) {
    cout << t << endl;
}

template <typename First, typename... Rest> void print(const First& first, const Rest&... rest) {
    cout << first << ", ";
    print(rest...); // recursive call using pack expansion syntax
}

int main()
{
    print(); // calls first overload, outputting only a newline
    print(1); // calls second overload

    // these call the third overload, the variadic template,
    // which uses recursion as needed.
    print(10, 20);
    print(100, 200, 300);
    print("first", 2, "third", 3.14159);
}
```

Saída

```
1
10, 20
100, 200, 300
first, 2, third, 3.14159
```

NOTE

A maioria das implementações que incorporam funções de modelo Variadic usam recursão de alguma forma, mas é ligeiramente diferente da recursão tradicional. A recursão tradicional envolve uma função que se chama usando a mesma assinatura. (Ela pode estar sobreescrita ou modelada, mas a mesma assinatura é escolhida a cada vez.) A recursão de Variadic envolve chamar um modelo de função Variadic usando diferentes números de argumentos (quase sempre diminuindo) e, portanto, carimbando uma assinatura diferente todas as vezes. Um "caso base" ainda é necessário, mas a natureza de recursão é diferente.

Expressões pós-fixadas

02/09/2020 • 11 minutes to read • [Edit Online](#)

As expressões pós-fixadas consistem em expressões primárias ou expressões nas quais operadores pós-fixados seguem uma expressão primária. Os operadores pós-fixados estão listados na tabela a seguir.

Operadores pós-fixados

NOME DO OPERADOR	NOTAÇÃO DO OPERADOR
Operador subscrito	[]
Operador de chamada de função	()
Operador de conversão de tipo explícita	<i>tipo-nome</i> ()
Operador de acesso de membro	. or ->
Operador de incremento pós-fixado	++
Operador de decremento pós-fixado	--

A sintaxe a seguir descreve expressões pós-fixadas possíveis:

```
primary-expression
postfix-expression[expression]postfix-expression(expression-list)simple-type-name(expression-list)postfix-
expression.namepostfix-expression->namepostfix-expression++postfix-expression--cast-keyword < typename >
(expression )typeid ( typename )
```

A *expressão de sufixo* acima pode ser uma [expressão primária](#) ou outra expressão de sufixo. As expressões pós-fixadas são agrupadas da esquerda para a direita; com isso, elas podem ser encadeadas da seguinte forma:

```
func(1)->GetValue()++
```

Na expressão acima, é uma expressão `func` primária, `func(1)` é uma expressão de sufixo de função, `func(1)->GetValue` é uma expressão de sufixo que especifica um membro da classe, `func(1)->GetValue()` é outra expressão de sufixo de função, e a expressão inteira é uma expressão de sufixo que aumenta o valor de retorno de `GetValue`. O significado da expressão, no conjunto, é "função de chamada passando 1 como um argumento e obtendo um ponteiro para uma classe como um valor de retorno". Em seguida `GetValue()`, chame essa classe e, em seguida, aumente o valor retornado.

As expressões listadas acima são expressões de atribuição, ou seja, o resultado dessas expressões deve ser um r-value.

O formato de expressão pós-fixada

```
simple-type-name ( expression-list )
```

indica a invocação do construtor. Caso `simple-type-name` seja um tipo fundamental, a lista de expressões deve ser formada por uma única expressão, a qual indica uma conversão do valor da expressão no tipo fundamental.

Esse tipo de expressão convertida imita um construtor. Como esse formato permite que classes e tipos fundamentais sejam construídos usando a mesma sintaxe, ele é especialmente útil na definição de classes de modelo.

A *palavra-chave Cast* é um de `dynamic_cast` , `static_cast` ou `reinterpret_cast` . Mais informações podem ser encontradas em `dynamic_cast` `static_cast` e `reinterpret_cast` .

O `typeid` operador é considerado uma expressão de sufixo. Consulte o operador `typeid`.

Argumentos formais e reais

Chamar programas transmite informações às funções chamadas em “argumentos reais”. As funções chamadas acessam as informações usando “argumentos formais” correspondentes.

Quando uma função for chamada, as seguintes tarefas serão executadas:

- Todos os argumentos reais (aqueles fornecidos pelo chamador) são avaliados. Não há nenhuma ordem implícita na qual esses argumentos são avaliados, mas todos os argumentos são avaliados e todos os efeitos colaterais são concluídos antes da entrada na função.
- Cada argumento formal é inicializado com seu argumento correspondente na lista de expressões. (Um argumento formal é um argumento declarado no cabeçalho da função e usado no corpo de uma função.) As conversões são feitas como se fosse por inicialização – as conversões padrão e definidas pelo usuário são executadas na conversão de um argumento real para o tipo correto. A inicialização executada é ilustrada de modo conceitual pelo seguinte código:

```
void Func( int i ); // Function prototype
...
Func( 7 );           // Execute function call
```

As inicializações conceituais antes da chamada são:

```
int Temp_i = 7;
Func( Temp_i );
```

A inicialização é executada como se estivesse usando a sintaxe de sinal de igualdade em vez de sintaxe de parênteses. Uma cópia de `i` é feita antes de transmitir o valor à função. (Para obter mais informações, consulte [inicializadores e conversões](#)).

Portanto, se o protótipo de função (declaração) chamar um argumento do tipo `long` e se o programa de chamada fornecer um argumento real do tipo `int` , o argumento real será promovido usando uma conversão de tipo padrão para tipo `long` (consulte [conversões padrão](#)).

É um erro fornecer um argumento real para o qual não há conversão padrão ou definida pelo usuário para o tipo de argumento formal.

Para argumentos reais de tipos de classe, o argumento formal é inicializado chamando o construtor da classe. (Consulte [construtores](#) para obter mais informações sobre essas funções de membros de classe especiais.)

- A chamada de função é executada.

O seguinte fragmento de programa demonstra uma chamada de função:

```

// expre_Formal_and_Actual_Arguments.cpp
void func( long param1, double param2 );

int main()
{
    long i = 1;
    double j = 2;

    // Call func with actual arguments i and j.
    func( i, j );
}

// Define func with formal parameters param1 and param2.
void func( long param1, double param2 )
{
}

```

Quando `func` é chamado de Main, o parâmetro formal `param1` é inicializado com o valor de `i` (`i` é convertido em Type `long` para corresponder ao tipo correto usando uma conversão padrão) e o parâmetro formal `param2` é inicializado com o valor de `j` (`j` é convertido para o tipo `double` usando uma conversão padrão).

Tratamento de tipos de argumento

Argumentos formais declarados como `const` tipos não podem ser alterados no corpo de uma função. As funções podem alterar qualquer argumento que não seja do tipo `const`. No entanto, a alteração é local para a função e não afeta o valor do argumento real, a menos que o argumento real tenha sido uma referência a um objeto que não seja do tipo `const`.

As seguintes funções ilustram alguns desses conceitos:

```

// expre_Treatment_of_Argument_Types.cpp
int func1( const int i, int j, char *c ) {
    i = 7;    // C3892 i is const.
    j = i;    // value of j is lost at return
    *c = 'a' + j;    // changes value of c in calling function
    return i;
}

double& func2( double& d, const char *c ) {
    d = 14.387;    // changes value of d in calling function.
    *c = 'a';    // C3892 c is a pointer to a const object.
    return d;
}

```

Reticências e argumentos padrão

As funções podem ser declaradas para aceitar menos argumentos do que o especificado na definição da função, usando um de dois métodos: reticências (`...`) ou argumentos padrão.

Reticências indica que os argumentos podem ser necessários, mas que o número e os tipos não são especificados na declaração. Essa geralmente é uma prática inadequada de programação do C++ porque ela elimina um dos benefícios de segurança do tipo C++: Conversões diferentes são aplicadas a funções declaradas com reticências do que às funções para as quais os tipos de argumento formal e real são conhecidos:

- Se o argumento real for do tipo `float`, ele será promovido para `double` o tipo antes da chamada de função.

- Qualquer `signed char` `unsigned char` campo ou, `signed short` ou `unsigned short`, tipo enumerado ou de bits é convertido em uma `signed int` promoção do ou do `unsigned int` usando integral.
- Qualquer argumento de tipo de classe é passado por valor como estrutura de dados; a cópia é criada por cópia binário em vez de chamar o construtor de cópia de classe (se houver).

Reticências, se usadas, devem ser declaradas por último na lista de argumentos. Para obter mais informações sobre como passar um número variável de argumentos, consulte a discussão sobre `va_arg`, `va_start` e `va_list` na *referência da biblioteca de tempo de execução*.

Para obter informações sobre argumentos padrão na programação CLR, consulte [listas de argumentos variáveis \(...\) \(C++/CLI\)](#).

Os argumentos padrão permitem que você especifique o valor que um argumento deverá assumir caso nenhum seja fornecido na chamada de função. O fragmento de código a seguir mostra como os argumentos padrão funcionam. Para obter mais informações sobre restrições na especificação de argumentos padrão, consulte [argumentos padrão](#).

```
// expre_Ellipsis_and_Default_Arguments.cpp
// compile with: /EHsc
#include <iostream>

// Declare the function print that prints a string,
// then a terminator.
void print( const char *string,
            const char *terminator = "\n" );

int main()
{
    print( "hello," );
    print( "world!" );

    print( "good morning", " " );
    print( "sunshine." );
}

using namespace std;
// Define print.
void print( const char *string, const char *terminator )
{
    if( string != NULL )
        cout << string;

    if( terminator != NULL )
        cout << terminator;
}
```

O programa anterior declara uma função, `print`, que usa dois argumentos. No entanto, o segundo argumento, `terminador`, tem um valor padrão, `"\n"`. No `main`, as duas primeiras chamadas para `print` permitir que o segundo argumento padrão forneça uma nova linha para encerrar a cadeia de caracteres impressa. A terceira chamada especifica um valor explícito para o segundo argumento. A saída do programa é

```
hello,
world!
good morning, sunshine.
```

Confira também

[Tipos de expressões](#)

Expressões com operadores unários

02/09/2020 • 2 minutes to read • [Edit Online](#)

Os operadores unários atuam somente em um operando em uma expressão. Os operadores unários são os seguintes:

- Operador de indireção (*)
- Operador de endereço (&)
- Operador unário de adição (+)
- Operador de negação unário (-)
- Operador de negação lógica (!)
- Operador de complemento de um (~)
- Operador de incremento de prefixo (++)
- Operador de diminuição de prefixo (--)
- Operador cast ()
- `sizeof` operador
- `__uuidof` operador
- `alignof` operador
- `new` operador
- `delete` operador

Esses operadores binários possuem associatividade da direita para a esquerda. As expressões unárias geralmente envolvem a sintaxe que precede uma expressão de sufixo ou primária.

As formas possíveis de expressões unárias são estas:

- *postfix-expression*
- `++` *expressão-unária*
- `--` *expressão-unária*
- *expressão de conversão de operador unário*
- `* sizeof` ****expressão unário*
- `sizeof(nome do tipo)`
- `decltype(expressão de)`
- *expressão de alocação*
- *expressão de desalocação*

Qualquer *expressão de sufixo* é considerada uma *expressão unária*, como qualquer expressão primária é considerada uma *expressão de sufixo*, todas as expressões primárias também são consideradas uma *expressão*

unário. Para obter mais informações, consulte [expressões de sufixo](#) e [expressões primárias](#).

Um *operador unário* consiste em um ou mais dos seguintes símbolos: `* & + - ! ~`

A *expressão CAST* é uma expressão unário com uma conversão opcional para alterar o tipo. Para obter mais informações, consulte [operador de conversão: \(\)](#).

Uma *expressão* pode ser qualquer expressão. Para obter mais informações, consulte [expressões](#).

A *expressão de alocação* refere-se ao `new` operador. A *expressão de desalocação* se refere ao `delete` operador. Para obter mais informações, consulte os links anteriores deste tópico.

Confira também

[Tipos de expressões](#)

Expressões com operadores binários

15/04/2020 • 2 minutes to read • [Edit Online](#)

Os operadores binários atuam em dois operandos em uma expressão. Os operadores binários são:

- [Operadores multiplicativos](#)

- Multiplicação (*)
 - Divisão (/)
 - Módulo (%)

- [Operadores aditivos](#)

- Adição (+)
 - Subtração (-)

- [Operadores shift](#)

- Turno certo (>>)
 - Turno esquerdo (<<)

- [Operadores relacionais e de igualdade](#)

- Menor que (<)
 - Maior que (>)
 - Menor ou igual < a (=)
 - Maior que ou igual a (>=)
 - Igual a (==)
 - Diferente de (!=)

- Operadores bit a bit

- [Bitwise E \(&\)](#)
 - [OR exclusivo bitwise \(^\)](#)
 - [BITWise inclusive OR \(|\)](#)

- Operadores lógicos

- [Lógica E \(&&\)](#)
 - [OR Lógico \(||\)](#)

- [Operadores de atribuição](#)

- Atribuição (=)
 - Atribuição de adição (+=)
 - Atribuição de subtração (-=)

- Atribuição de multiplicação (*=)
- Atribuição de divisão (/=)
- Atribuição de módulo (%=)
- Atribuição de <turno esquerdo (<=)
- Atribuição de turno direito (> >=)
- Bitwise E atribuição (&=)
- Atribuição OR exclusivo de bit a bit (^=)
- Atribuição ou ou inclusiva bitwise (|=)
- [Operador de Comma \(,\)](#)

Confira também

[Tipos de expressões](#)

Expressões de constante C++

02/09/2020 • 2 minutes to read • [Edit Online](#)

Um valor *constante* é aquele que não é alterado. O C++ fornece duas palavras-chave para permitir que você expresse a intenção de que um objeto não se destina a ser modificado e para impor essa intenção.

O C++ requer expressões constantes — expressões que são avaliadas como uma constante — para declarações de:

- Limites de matriz
- Seletores em instruções `case`
- Especificação de comprimento do campo de bits
- Inicializadores de enumeração

Os únicos operandos que são válidos em expressões constantes são:

- Literais
- Constantes de enumeração
- Valores declarados como `const` que são inicializados com expressões constantes
- `sizeof` expressões

As constantes não integral devem ser convertidas (explicitamente ou implicitamente) em tipos inteiros para serem válidas em uma expressão constante. Portanto, o código a seguir é válido:

```
const double Size = 11.0;
char chArray[(int)Size];
```

Conversões explícitas para tipos inteiros são expressões de constantes legais; todos os outros tipos e tipos derivados são ilegais, exceto quando usados como operandos para o `sizeof` operador.

O operador vírgula e os operadores de atribuição não podem ser usados em expressões constantes.

Confira também

[Tipos de expressões](#)

Semântica de expressões

02/09/2020 • 8 minutes to read • [Edit Online](#)

As expressões são avaliadas de acordo com a precedência e o agrupamento dos respectivos operadores.

([Precedência de operador e Associação](#) em [convenções lexicais](#), mostra as relações que os operadores do C++ impõem em expressões.)

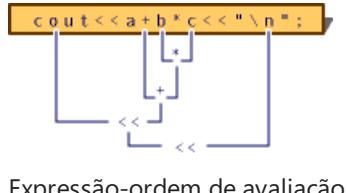
Ordem de avaliação

Considere este exemplo:

```
// Order_of_Evaluation.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;
int main()
{
    int a = 2, b = 4, c = 9;

    cout << a + b * c << "\n";
    cout << a + (b * c) << "\n";
    cout << (a + b) * c << "\n";
}
```

```
38
38
54
```

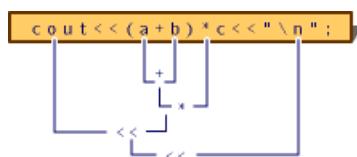


Expressão-ordem de avaliação

A ordem na qual a expressão mostrada na figura acima é avaliada é determinada pela precedência e associatividade dos operadores:

1. A multiplicação (*) tem a precedência mais alta nessa expressão; portanto, a subexpressão `b * c` é avaliada primeiro.
2. A adição (+) tem a precedência mais alta a seguir, de modo que `a` é adicionado ao produto de `b` vezes `c`.
3. SHIFT esquerda (<<) tem a menor precedência na expressão, mas há duas ocorrências. Como o operador de deslocamento para a esquerda é agrupado da esquerda para a direita, a subexpressão à esquerda é avaliada primeiro, seguida pela subexpressão à direita.

Quando são usados parênteses para agrupar as subexpressões, eles alteram a precedência e também a ordem em que a expressão é avaliada, conforme mostra a figura a seguir.



Expressão-ordem de avaliação com parênteses

Expressões como as da figura acima são avaliadas apenas por seus efeitos colaterais — nesse caso, para transferir informações para o dispositivo de saída padrão.

Notação em expressões

A linguagem C++ especifica determinadas compatibilidades ao especificar operandos. A tabela a seguir mostra os tipos de operandos aceitáveis para operadores que exigem operandos *do tipo Type*.

Tipos de operandos aceitáveis para operadores

TIPO ESPERADO	TIPOS PERMITIDOS
<i>tipo</i>	* <code>const</code> *** <i>tipo</i> de * <code>volatile</code> *** <i>tipo</i> de <i>Escreva&</i> * <code>const</code> *** <i>tipo</i> de& * <code>volatile</code> *** <i>tipo</i> de& <code>volatile const</code> <i>tipo</i> de <code>volatile const</code> <i>tipo</i> de&
<i>tipo</i> de*	<i>tipo</i> de* * <code>const</code> *** <i>tipo</i> de* * <code>volatile</code> *** <i>tipo</i> de* <code>volatile const</code> <i>tipo</i> de*
* <code>const</code> *** <i>tipo</i> de	<i>tipo</i> * <code>const</code> *** <i>tipo</i> de * <code>const</code> *** <i>tipo</i> de&
* <code>volatile</code> *** <i>tipo</i> de	<i>tipo</i> * <code>volatile</code> *** <i>tipo</i> de * <code>volatile</code> *** <i>tipo</i> de&

Como as regras acima podem sempre ser usadas em conjunto, um ponteiro const para um objeto volatile pode ser fornecido onde um ponteiro é esperado.

Expressões ambíguas

Algumas expressões são ambíguas em seu significado. Essas expressões ocorrem com mais frequência quando o valor de um objeto é modificado mais de uma vez na mesma expressão. Essas expressões confiam em uma determinada ordem de avaliação, onde a linguagem não define uma. Considere o exemplo a seguir:

```
int i = 7;  
  
func( i, ++i );
```

A linguagem C++ não garante a ordem de avaliação dos argumentos para uma chamada de função. Portanto, no exemplo anterior, `func` pode receber os valores 7 e 8 ou 8 e 8 para seus parâmetros, dependendo se os parâmetros são avaliados da esquerda para a direita ou da direita para a esquerda.

Pontos de sequência do C++ (específicos da Microsoft)

Uma expressão pode modificar o valor de um objeto apenas uma vez entre "pontos de sequência" consecutivos.

No momento, a definição da linguagem C++ não especifica pontos de sequência. O Microsoft C++ usa os mesmos pontos de sequência que o ANSI C para qualquer expressão que envolva operadores de C e não envolva operadores sobrecarregados. Quando os operadores estão sobrecarregados, a semântica muda de sequenciamento de operadores para sequenciamento de chamada de funções. O Microsoft C++ os seguintes pontos de sequência:

- Operando esquerdo do operador AND lógico (`&&`). O operando esquerdo do operador AND lógico é completamente avaliado e todos os efeitos colaterais são concluídos antes de continuar. Não há nenhuma garantia de que o operando direito do operador AND lógico será avaliado.
- Operando esquerdo do operador OR lógico (`||`). O operando esquerdo do operador OR lógico é completamente avaliado e todos os efeitos colaterais são concluídos antes de continuar. Não há nenhuma garantia de que o operando direito do operador OR lógico será avaliado.
- Operando esquerdo do operador vírgula. O operando esquerdo do operador vírgula é completamente avaliado e todos os efeitos colaterais são concluídos antes de continuar. Os dois operandos do operador vírgula são sempre avaliados.
- Operador de chamada de função. A expressão de chamada de função e todos os argumentos para uma função, inclusive os argumentos padrão, são avaliados e todos os efeitos colaterais são concluídos antes da entrada na função. Não há nenhuma ordem de avaliação especificada entre os argumentos ou a expressão de chamada de função.
- Primeiro operando do operador condicional. O primeiro operando do operador condicional é completamente avaliado e todos os efeitos colaterais são concluídos antes de continuar.
- O término de uma expressão de inicialização completa, por exemplo, o término de uma inicialização em uma instrução de declaração.
- A expressão em uma instrução de expressão. As instruções da expressão consistem em uma expressão opcional seguida por um ponto e vírgula (`;`). A expressão é completamente avaliada em relação a seus efeitos colaterais.
- A expressão de controle em uma instrução de seleção (`if` ou `switch`). A expressão é completamente avaliada e todos os efeitos colaterais são concluídos antes que o código dependente da seleção seja executado.
- A expressão de controle de uma instrução `while` ou `do`. A expressão é completamente avaliada e todos os efeitos colaterais são concluídos antes que as instruções na próxima iteração do loop `while` ou `do` sejam executadas.
- Cada uma das três expressões de uma instrução `for`. Cada expressão é completamente avaliada e todos os efeitos colaterais são concluídos antes de passar para a próxima expressão.
- A expressão em uma instrução `return`. A expressão é completamente avaliada e todos os efeitos colaterais são concluídos antes que o controle retorne à função de chamada.

Confira também

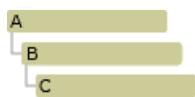
[Expressões](#)

Conversão

25/03/2020 • 3 minutes to read • [Edit Online](#)

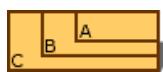
A linguagem C++ estabelece que se uma classe é derivada de uma classe base que contém funções virtuais, um ponteiro para esse tipo de classe base pode ser usado para chamar as implementações das funções virtuais que residem no objeto de classe derivada. Uma classe que contém funções virtuais é às vezes chamada de "classe polimórfica".

Como uma classe derivada contém completamente as definições de todas as classes base de que é derivada, é seguro converter um ponteiro superior da hierarquia de classes em qualquer uma dessas classes base. Dado um ponteiro para uma classe base, talvez seja seguro converter o ponteiro abaixo na hierarquia. É seguro se o objeto que está sendo apontado seja realmente de um tipo derivado da classe base. Nesse caso, o objeto real é considerado o "objeto completo". O ponteiro para classe base é considerado um "subobjeto" do objeto completo. Por exemplo, considere a hierarquia da classe mostrada na figura a seguir.



Hierarquia de classe

Um objeto do tipo `C` pode ser visualizado conforme mostrado na figura a seguir.



Classe C com subobjetos B e um

Dada uma instância da classe `C`, há um subobjeto `B` e um subobjeto `A`. A instância de `C`, inclusive os subobjetos `A` e `B`, é o "objeto completo".

Usando as informações de tipo de tempo de execução, é possível verificar se um ponteiro aponta realmente para um objeto completo e pode ser convertido seguramente para apontar para outro objeto em sua hierarquia. O operador `dynamic_cast` pode ser usado para fazer esses tipos de conversões. Também executa a verificação de tempo de execução necessária para tornar a operação segura.

Para a conversão de tipos nonpolymorphic, você pode usar o operador de `static_cast` (este tópico explica a diferença entre conversões de conversão estática e dinâmica e quando é apropriado usar cada uma).

Esta seção contém os seguintes tópicos:

- [Operadores de conversão](#)
- [Informações de tipo de tempo de execução](#)

Confira também

[Expressões](#)

Operadores de conversão

02/09/2020 • 2 minutes to read • [Edit Online](#)

Há vários operadores de conversão específicos à linguagem C++. Esses operadores são destinados a remover qualquer ambiguidade e perigo inerente no estilo antigo de conversões da linguagem C. Esses operadores são:

- [dynamic_cast](#) Usado para conversão de tipos polimórficos.
- [static_cast](#) Usado para conversão de tipos nonpolymorphic.
- [const_cast](#) Usado para remover os `const` `volatile` atributos, e `__unaligned`.
- [reinterpret_cast](#) Usado para reinterpretAÇÃO simples de bits.
- [safe_cast](#) Usado em C++/CLI para produzir MSIL verificável.

Use `const_cast` e `reinterpret_cast` como último recurso, já que esses operadores apresentam os mesmos perigos que as conversões de estilo antigas. No entanto, ainda são necessários para substituir completamente as conversões antigas.

Confira também

[Conversão](#)

Operador dynamic_cast

02/09/2020 • 11 minutes to read • [Edit Online](#)

Converte o operando `expression` em um objeto do tipo `type-id`.

Sintaxe

```
dynamic_cast < type-id > ( expression )
```

Comentários

`type-id` deve ser um ponteiro ou uma referência a um tipo previamente definido da classe ou a um "ponteiro para nulo". O tipo de `expression` deve ser um ponteiro se `type-id` for um ponteiro, ou um l-value se `type-id` for uma referência.

Consulte [static_cast](#) para obter uma explicação da diferença entre as conversões de conversão estática e dinâmica e quando é apropriado usar cada uma delas.

Há duas alterações significativas no comportamento do `dynamic_cast` em código gerenciado:

- `dynamic_cast` para um ponteiro para o tipo subjacente de uma enumeração Boxed falhará em tempo de execução, retornando 0 em vez do ponteiro convertido.
- `dynamic_cast` não gerará mais uma exceção quando `type-id` for um ponteiro interior para um tipo de valor, com a falha de conversão em tempo de execução. Agora, a conversão retornará ao valor 0 do ponteiro ao invés de gerar.

Se `type-id` for um ponteiro para uma classe base direta ou indireta, inequívoca e acessível de `expression`, um ponteiro para o subobjeto exclusivo do tipo `type-id` é o resultado. Por exemplo:

```
// dynamic_cast_1.cpp
// compile with: /c
class B {};
class C : public B {};
class D : public C {};

void f(D* pd) {
    C* pc = dynamic_cast<C*>(pd);    // ok: C is a direct base class
                                         // pc points to C subobject of pd
    B* pb = dynamic_cast<B*>(pd);    // ok: B is an indirect base class
                                         // pb points to B subobject of pd
}
```

Esse tipo de conversão é chamada de "upcast", pois move um ponteiro para uma hierarquia da classe acima, de uma classe derivada para uma classe de derivação. Um upcast é uma conversão implícita.

Se `type-id` é nulo*, uma verificação de tempo de execução será feita para determinar o tipo real de `expression`. O resultado é um ponteiro para o objeto completo apontado por `expression`. Por exemplo:

```

// dynamic_cast_2.cpp
// compile with: /c /GR
class A {virtual void f();};
class B {virtual void f();};

void f() {
    A* pa = new A;
    B* pb = new B;
    void* pv = dynamic_cast<void*>(pa);
    // pv now points to an object of type A

    pv = dynamic_cast<void*>(pb);
    // pv now points to an object of type B
}

```

Se `type-id` não é nulo*, uma verificação de tempo de execução será feita para verificar se o objeto apontado por `expression` pode ser convertido para o tipo apontado por `type-id`.

Se o tipo de `expression` é uma classe base do tipo `type-id`, uma verificação de tempo de execução será feita para verificar se `expression` realmente aponta para um objeto completo do tipo `type-id`. Se isso ocorrer, o resultado é um ponteiro para um objeto completo do tipo `type-id`. Por exemplo:

```

// dynamic_cast_3.cpp
// compile with: /c /GR
class B {virtual void f();};
class D : public B {virtual void f();};

void f() {
    B* pb = new D;    // unclear but ok
    B* pb2 = new B;

    D* pd = dynamic_cast<D*>(pb);    // ok: pb actually points to a D
    D* pd2 = dynamic_cast<D*>(pb2);    // pb2 points to a B not a D
}

```

Esse tipo de conversão é chamada de "downcast", pois move um ponteiro para uma hierarquia da classe abaixo, de uma classe determinada para uma classe derivada dela.

Em casos de herança múltipla, as possibilidades de ambiguidade são introduzidas. Considere a hierarquia da classe mostrada na figura a seguir.

Para tipos CLR, `dynamic_cast` resulta em um não op se a conversão puder ser executada implicitamente ou uma `isinst` instrução MSIL, que executa uma verificação dinâmica e retorna `nullptr` se a conversão falhar.

O exemplo a seguir usa `dynamic_cast` para determinar se uma classe é uma instância de um tipo específico:

```

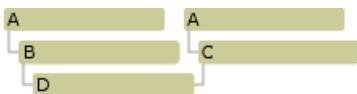
// dynamic_cast_clr.cpp
// compile with: /clr
using namespace System;

void PrintObjectType( Object^o ) {
    if( dynamic_cast<String^>(o) )
        Console::WriteLine("Object is a String");
    else if( dynamic_cast<int^>(o) )
        Console::WriteLine("Object is an int");
}

int main() {
    Object^o1 = "hello";
    Object^o2 = 10;

    PrintObjectType(o1);
    PrintObjectType(o2);
}

```



Hierarquia de classes que mostra várias heranças

Um ponteiro para um objeto do tipo `D` pode seguramente ser gerado em `B` ou `C`. No entanto, se `D` for gerado para apontar para um objeto `A`, qual instância de `A` resultaria? Isso resultará em um erro ambíguo de geração. Para contornar esse problema, você pode executar duas conversões inequívocas. Por exemplo:

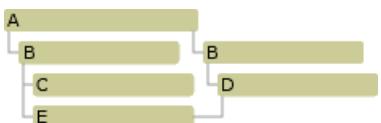
```

// dynamic_cast_4.cpp
// compile with: /c /GR
class A {virtual void f();};
class B : public A {virtual void f();};
class C : public A {virtual void f();};
class D : public B, public C {virtual void f();};

void f() {
    D* pd = new D;
    A* pa = dynamic_cast<A*>(pd);    // C4540, ambiguous cast fails at runtime
    B* pb = dynamic_cast<B*>(pd);    // first cast to B
    A* pa2 = dynamic_cast<A*>(pb);    // ok: unambiguous
}

```

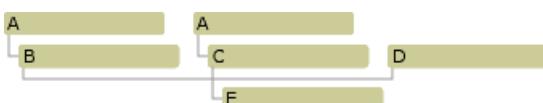
As ambiguidades adicionais podem ser introduzidas quando você usar classes base virtuais. Considere a hierarquia da classe mostrada na figura a seguir.



Hierarquia de classes que mostra as classes base virtuais

Nesta hierarquia, `A` é uma classe base virtual. Dada uma instância da classe `E` e um ponteiro para o `A` subobjeto, um `dynamic_cast` para um ponteiro para `B` falhará devido à ambiguidade. Primeiro você deve converter de volta ao objeto completo `E`, então trabalhar até a hierarquia, de maneira não ambígua, para alcançar o objeto correto `B`.

Considere a hierarquia da classe mostrada na figura a seguir.



Hierarquia de classes que mostra classes base duplicadas

Dado um objeto de tipo `E` e um ponteiro para o subobjeto `D`, para navegar do subobjeto `D` ao subobjeto mais à esquerda `A`, três conversões podem ser feitas. Você pode executar uma `dynamic_cast` conversão do `D` ponteiro para um `E` ponteiro, depois uma conversão (`dynamic_cast` ou uma conversão implícita) de `E` para `B`, finalmente, uma conversão implícita de `B` para `A`. Por exemplo:

```
// dynamic_cast_5.cpp
// compile with: /c /GR
class A {virtual void f();};
class B : public A {virtual void f();};
class C : public A {};
class D {virtual void f();};
class E : public B, public C, public D {virtual void f();};

void f(D* pd) {
    E* pe = dynamic_cast<E*>(pd);
    B* pb = pe;    // upcast, implicit conversion
    A* pa = pb;    // upcast, implicit conversion
}
```

O `dynamic_cast` operador também pode ser usado para executar uma "conversão cruzada". Usando a mesma hierarquia da classe, é possível converter um ponteiro, por exemplo, do subobjeto `B` ao para o subobjeto `D`, contanto que o objeto completo seja do tipo `E`.

Considerando as conversões cruzadas, é realmente possível fazer a conversão de um ponteiro para `D` para um ponteiro para o subobjeto mais à esquerda de `A` em apenas duas etapas. Você pode executar uma conversão cruzada de `D` para `B`, e uma conversão implícita de `B` para `A`. Por exemplo:

```
// dynamic_cast_6.cpp
// compile with: /c /GR
class A {virtual void f();};
class B : public A {virtual void f();};
class C : public A {};
class D {virtual void f();};
class E : public B, public C, public D {virtual void f();};

void f(D* pd) {
    B* pb = dynamic_cast<B*>(pd);    // cross cast
    A* pa = pb;    // upcast, implicit conversion
}
```

Um valor de ponteiro nulo é convertido para o valor de ponteiro nulo do tipo de destino por `dynamic_cast`.

Quando você usa `dynamic_cast < type-id > (expression)`, se `expression` não puder ser convertido com segurança para o tipo `type-id`, a verificação de tempo de execução fará com que a conversão falhe. Por exemplo:

```
// dynamic_cast_7.cpp
// compile with: /c /GR
class A {virtual void f();};
class B {virtual void f();};

void f() {
    A* pa = new A;
    B* pb = dynamic_cast<B*>(pa);    // fails at runtime, not safe;
    // B not derived from A
}
```

O valor de uma conversão falhada para o tipo de ponteiro é o ponteiro nulo. Um tipo de referência de conversão com falha gera uma [exceção de bad_cast](#). Se `expression` o não apontar para ou referenciar um objeto válido, uma `__non_rtti_object` exceção será lançada.

Consulte [typeid](#) para obter uma explicação da `__non_rtti_object` exceção.

Exemplo

O exemplo a seguir cria o ponteiro (struct A) da classe base, para um objeto (struct C). Isso, além da existência de funções virtuais, permite a polimorfismo de runtime.

O exemplo também chama uma função não virtual na hierarquia.

```

// dynamic_cast_8.cpp
// compile with: /GR /EHsc
#include <stdio.h>
#include <iostream>

struct A {
    virtual void test() {
        printf_s("in A\n");
    }
};

struct B : A {
    virtual void test() {
        printf_s("in B\n");
    }

    void test2() {
        printf_s("test2 in B\n");
    }
};

struct C : B {
    virtual void test() {
        printf_s("in C\n");
    }

    void test2() {
        printf_s("test2 in C\n");
    }
};

void Globaltest(A& a) {
    try {
        C &c = dynamic_cast<C&>(a);
        printf_s("in GlobalTest\n");
    }
    catch(std::bad_cast) {
        printf_s("Can't cast to C\n");
    }
}

int main() {
    A *pa = new C;
    A *pa2 = new B;

    pa->test();

    B * pb = dynamic_cast<B *>(pa);
    if (pb)
        pb->test2();

    C * pc = dynamic_cast<C *>(pa2);
    if (pc)
        pc->test2();

    C ConStack;
    Globaltest(ConStack);

    // will fail because B knows nothing about C
    B BonStack;
    Globaltest(BonStack);
}

```

```
in C
test2 in B
in GlobalTest
Can't cast to C
```

Confira também

[Operadores de conversão](#)

[Palavras-chave](#)

Exceção bad_cast

02/09/2020 • 2 minutes to read • [Edit Online](#)

A exceção **bad_cast** é gerada pelo `dynamic_cast` operador como o resultado de uma conversão com falha para um tipo de referência.

Sintaxe

```
catch (bad_cast)
    statement
```

Comentários

A interface para **bad_cast** é:

```
class bad_cast : public exception
```

O código a seguir contém um exemplo de um falha `dynamic_cast` que gera a exceção **bad_cast** .

```
// expre_bad_cast_Exception.cpp
// compile with: /EHsc /GR
#include <typeinfo>
#include <iostream>

class Shape {
public:
    virtual void virtualfunc() const {}
};

class Circle: public Shape {
public:
    virtual void virtualfunc() const {}
};

using namespace std;
int main() {
    Shape shape_instance;
    Shape& ref_shape = shape_instance;
    try {
        Circle& ref_circle = dynamic_cast<Circle&>(ref_shape);
    }
    catch (bad_cast b) {
        cout << "Caught: " << b.what();
    }
}
```

A exceção é gerada porque o objeto que está sendo convertido (uma forma) não é derivado do tipo de conversão especificado (Circle). Para evitar a exceção, adicione estas declarações a `main` :

```
Circle circle_instance;
Circle& ref_circle = circle_instance;
```

Em seguida, inverta a noção da conversão no `try` bloco da seguinte maneira:

```
Shape& ref_shape = dynamic_cast<Shape&>(ref_circle);
```

Membros

Construtores

CONSTRUTOR	DESCRIÇÃO
<code>bad_cast</code>	O construtor para objetos do tipo <code>bad_cast</code> .

Funções

FUNÇÃO	DESCRIÇÃO
<code>acontece</code>	TBD

Operadores

OPERADOR	DESCRIÇÃO
<code>operator =</code>	Um operador de atribuição que atribui um <code>bad_cast</code> objeto a outro.

bad_cast

O construtor para objetos do tipo `bad_cast`.

```
bad_cast(const char * _Message = "bad cast");  
bad_cast(const bad_cast &);
```

operador =

Um operador de atribuição que atribui um `bad_cast` objeto a outro.

```
bad_cast& operator=(const bad_cast&) noexcept;
```

acontece

```
const char* what() const noexcept override;
```

Confira também

[Operador de `dynamic_cast`](#)

[Palavras-chave](#)

[Práticas recomendadas do C++ moderno para exceções e tratamento de erros](#)

Operador static_cast

02/09/2020 • 7 minutes to read • [Edit Online](#)

Converte uma *expressão* para o tipo de *ID-tipo*, com base apenas nos tipos que estão presentes na expressão.

Sintaxe

```
static_cast <type-id> ( expression )
```

Comentários

No padrão C++, nenhuma verificação de tipo de tempo de execução é feita para ajudar a garantir a segurança da conversão. No C++/CX, uma verificação de tempo de compilação e de runtime é executada. Para obter mais informações, consulte [Conversão](#).

O `static_cast` operador pode ser usado para operações como converter um ponteiro para uma classe base em um ponteiro para uma classe derivada. Essas conversões não são sempre seguras.

Em geral, você usa `static_cast` quando deseja converter tipos de dados numéricos, como enums para ints ou ints como floats, e você tem certeza dos tipos de dados envolvidos na conversão. `static_cast` as conversões não são tão seguras quanto as `dynamic_cast` conversões, pois `static_cast` não há verificação de tipo em tempo de execução, enquanto `dynamic_cast` o faz. R `dynamic_cast` para um ponteiro ambíguo falhará, enquanto um `static_cast` retorna como se nada estivesse errado, isso pode ser perigoso. Embora `dynamic_cast` as conversões sejam mais seguras, `dynamic_cast` só funciona em ponteiros ou referências, e a verificação de tipo em tempo de execução é uma sobrecarga. Para obter mais informações, consulte [operador de dynamic_cast](#).

No exemplo a seguir, a linha `D* pd2 = static_cast<D*>(pb);` não é segura porque `D` pode ter campos e métodos que não estão em `B`. No entanto, a linha `B* pb2 = static_cast<B*>(pd);` é uma conversão segura porque `D` sempre contém tudo de `B`.

```
// static_cast_Operator.cpp
// compile with: /LD
class B {};

class D : public B {};

void f(B* pb, D* pd) {
    D* pd2 = static_cast<D*>(pb);    // Not safe, D can have fields
                                         // and methods that are not in B.

    B* pb2 = static_cast<B*>(pd);    // Safe conversion, D always
                                         // contains all of B.
}
```

Em contraste com `dynamic_cast`, nenhuma verificação de tempo de execução é feita na `static_cast` conversão de `pb`. O objeto apontado por `pb` não pode ser um objeto do tipo `D`, pois, nesse caso, o uso de `*pd2` seria desastroso. Por exemplo, chamar uma função que é membro da classe `D`, mas não da classe `B`, poderá resultar em uma violação de acesso.

Os `dynamic_cast` `static_cast` operadores e movem um ponteiro em toda a hierarquia de classe. No entanto,

`static_cast` depende exclusivamente das informações fornecidas na instrução `cast` e, portanto, pode não ser segura. Por exemplo:

```
// static_cast_Operator_2.cpp
// compile with: /LD /GR
class B {
public:
    virtual void Test(){}
};

class D : public B {};

void f(B* pb) {
    D* pd1 = dynamic_cast<D*>(pb);
    D* pd2 = static_cast<D*>(pb);
}
```

Se `pb` apontar para um objeto do tipo `D`, `pd1` e `pd2` obterão o mesmo valor. Eles também obterão o mesmo valor se `pb == 0`.

Se `pb` apontar para um objeto do tipo `B` e não para a `D` classe completa, então `dynamic_cast` saberá o suficiente para retornar zero. No entanto, `static_cast` só se baseia na declaração do programador que `pb` aponta para um objeto do tipo `D` e simplesmente retorna um ponteiro para o `D` objeto esperado.

Consequentemente, o `static_cast` pode fazer o inverso de conversões implícitas, caso em que os resultados são indefinidos. Ele é deixado ao programador para verificar se os resultados de uma `static_cast` conversão são seguros.

Esse comportamento também se aplica a tipos diferentes dos tipos de classe. Por exemplo, `static_cast` pode ser usado para converter de um `int` para um `char`. No entanto, o resultado `char` pode não ter bits suficientes para manter o `int` valor inteiro. Novamente, é deixada para o programador verificar se os resultados de uma `static_cast` conversão são seguros.

O `static_cast` operador também pode ser usado para executar qualquer conversão implícita, incluindo conversões padrão e conversões definidas pelo usuário. Por exemplo:

```
// static_cast_Operator_3.cpp
// compile with: /LD /GR
typedef unsigned char BYTE;

void f() {
    char ch;
    int i = 65;
    float f = 2.5;
    double dbl;

    ch = static_cast<char>(i);    // int to char
    dbl = static_cast<double>(f);  // float to double
    i = static_cast<BYTE>(ch);
}
```

O `static_cast` operador pode converter explicitamente um valor integral em um tipo de enumeração. Se o valor do tipo integral não estiver dentro do intervalo de valores de enumeração, o valor de enumeração resultante será indefinido.

O `static_cast` operador converte um valor de ponteiro nulo para o valor de ponteiro nulo do tipo de destino.

Qualquer expressão pode ser convertida explicitamente para o tipo `void` pelo `static_cast` operador. O tipo `void` de destino pode, opcionalmente `const`, incluir o `volatile` atributo, ou `__unaligned`.

O `static_cast` operador não pode converter os `const` `volatile` atributos, ou `__unaligned`. Consulte [operador de `const_cast`](#) para obter informações sobre como remover esses atributos.

C++/CLI: Devido ao perigo de executar conversões não verificadas sobre um coletor de lixo realocado, o uso do `static_cast` deve estar apenas no código de desempenho crítico quando você tiver certeza de que ele funcionará corretamente. Se você precisar usar `static_cast` o no modo de versão, substitua-o por `safe_cast` em suas compilações de depuração para garantir o sucesso.

Confira também

[Operadores de conversão](#)

[Palavras-chave](#)

Operador `const_cast`

02/09/2020 • 2 minutes to read • [Edit Online](#)

Remove os `const` `volatile` atributos, e `__unaligned` de uma classe.

Sintaxe

```
const_cast <type-id> (expression)
```

Comentários

Um ponteiro para qualquer tipo de objeto ou um ponteiro para um membro de dados pode ser explicitamente convertido em um tipo que seja idêntico, exceto para os `const` `volatile` `__unaligned` qualificadores, e. Para ponteiros e referências, o resultado fará referência ao objeto original. Para ponteiros para membros de dados, o resultado fará referência ao mesmo membro que o ponteiro original (não convertido) para o membro de dados. Dependendo do tipo do objeto referenciado, uma operação de gravação pelo ponteiro, referência ou ponteiro para o membro de dados resultante pode gerar comportamento indefinido.

Você não pode usar o `const_cast` operador para substituir diretamente o status constante de uma variável constante.

O `const_cast` operador converte um valor de ponteiro nulo para o valor de ponteiro nulo do tipo de destino.

Exemplo

```
// expe_const_cast_Operator.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
class CCTest {
public:
    void setNumber( int );
    void printNumber() const;
private:
    int number;
};

void CCTest::setNumber( int num ) { number = num; }

void CCTest::printNumber() const {
    cout << "\nBefore: " << number;
    const_cast< CCTest * >( this )->number--;
    cout << "\nAfter: " << number;
}

int main() {
    CCTest X;
    X.setNumber( 8 );
    X.printNumber();
}
```

Na linha que contém o `const_cast`, o tipo de dados do `this` ponteiro é `const CCTest *`. O `const_cast`

operador altera o tipo de dados do `this` ponteiro para `CCTest *`, permitindo que o membro `number` seja modificado. A conversão só durará pelo restante da instrução em que aparece.

Confira também

[Operadores de conversão](#)

[Palavras-chave](#)

Operador reinterpret_cast

02/09/2020 • 2 minutes to read • [Edit Online](#)

Permite que qualquer ponteiro seja convertido em outro tipo de ponteiro. Também permite que qualquer tipo integral seja convertido em qualquer tipo de ponteiro e vice-versa.

Sintaxe

```
reinterpret_cast < type-id > ( expression )
```

Comentários

O uso indevido do `reinterpret_cast` operador pode ser facilmente inseguro. A menos que a conversão desejada seja inherentemente de nível baixo, você deve usar um dos outros operadores de conversão.

O `reinterpret_cast` operador pode ser usado para conversões, como `char*` para `int*`, ou `One_class*` para `Unrelated_class*`, que são inherentemente inseguras.

O resultado de um `reinterpret_cast` não pode ser usado com segurança para qualquer coisa que não seja a conversão de volta para seu tipo original. Outros usos são, na melhor das hipóteses, não portáteis.

O `reinterpret_cast` operador não pode converter os `const` `volatile` atributos, ou `__unaligned`. Consulte [operador de const_cast](#) para obter informações sobre como remover esses atributos.

O `reinterpret_cast` operador converte um valor de ponteiro nulo para o valor de ponteiro nulo do tipo de destino.

Um uso prático do `reinterpret_cast` é em uma função de hash, que mapeia um valor para um índice de forma que dois valores distintos raramente terminem com o mesmo índice.

```

#include <iostream>
using namespace std;

// Returns a hash code based on an address
unsigned short Hash( void *p ) {
    unsigned int val = reinterpret_cast<unsigned int>( p );
    return ( unsigned short )( val ^ (val >> 16));
}

using namespace std;
int main() {
    int a[20];
    for ( int i = 0; i < 20; i++ )
        cout << Hash( a + i ) << endl;
}

```

Output:

```

64641
64645
64889
64893
64881
64885
64873
64877
64865
64869
64857
64861
64849
64853
64841
64845
64833
64837
64825
64829

```

O `reinterpret_cast` permite que o ponteiro seja tratado como um tipo integral. O resultado de bit é deslocado e recebe XOR para gerar um índice exclusivo (exclusivo para um alto nível de probabilidade). O índice é truncado em seguida por uma conversão padrão do estilo C para o tipo de retorno de função.

Confira também

[Operadores de conversão](#)

[Palavras-chave](#)

Informações de tipo de tempo de execução

02/09/2020 • 2 minutes to read • [Edit Online](#)

As informações de tipo em tempo de execução (RTTI) são um mecanismo que permite que o tipo de um objeto seja determinado durante a execução do programa. A RTTI foi adicionada à linguagem C++ porque muitos fornecedores de bibliotecas de classes implementavam essa funcionalidade de maneira independente. Isso causou incompatibilidades entre as bibliotecas. Assim, ficou claro que era necessário o suporte a informações de tipo em tempo de execução no nível da linguagem.

Para uma questão de clareza, esta discussão sobre a RTTI é quase que totalmente restrita a ponteiros. No entanto, os conceitos abordados também se aplicam a referências.

Há três principais elementos de linguagem C++ para as informações de tipo em tempo de execução:

- O operador de [dynamic_cast](#) .

Usado para a conversão de tipos polimorfos.

- O operador [typeid](#) .

Usado para identificar o tipo exato de um objeto.

- A classe [type_info](#) .

Usado para armazenar as informações de tipo retornadas pelo `typeid` operador.

Confira também

[Conversão](#)

Exceção bad_typeid

02/09/2020 • 2 minutes to read • [Edit Online](#)

A exceção **bad_typeid** é gerada pelo operador `typeid` quando o operando para `typeid` é um ponteiro NULL.

Sintaxe

```
catch (bad_typeid)
    statement
```

Comentários

A interface para **bad_typeid** é:

```
class bad_typeid : public exception
{
public:
    bad_typeid();
    bad_typeid(const char * _Message = "bad typeid");
    bad_typeid(const bad_typeid &);
    virtual ~bad_typeid();

    bad_typeid& operator=(const bad_typeid&);

    const char* what() const;
};
```

O exemplo a seguir mostra o `typeid` operador que gera uma exceção de **bad_typeid** .

```
// expre_bad_typeid.cpp
// compile with: /EHsc /GR
#include <typeinfo>
#include <iostream>

class A{
public:
    // object for class needs vtable
    // for RTTI
    virtual ~A();
};

using namespace std;
int main() {
A* a = NULL;

try {
    cout << typeid(*a).name() << endl; // Error condition
}
catch (bad_typeid){
    cout << "Object is NULL" << endl;
}
}
```

Saída

Object is NULL

Confira também

[Informações de tipo de tempo de execução](#)

[Palavras-chave](#)

Classe type_info

25/03/2020 • 4 minutes to read • [Edit Online](#)

A classe `type_info` descreve informações de tipo geradas dentro do programa pelo compilador. Os objetos dessa classe armazenam efetivamente um ponteiro para um nome do tipo. A classe `type_info` também armazena um valor codificado adequado para comparar dois tipos para a ordem de igualdade ou de agrupamento. As regras e a sequência de agrupamento de codificação para tipos não são especificados e podem ser diferentes entre os programas.

O arquivo de cabeçalho de `<typeinfo>` deve ser incluído para usar a classe `type_info`. A interface para a classe `type_info` é:

```
class type_info {
public:
    type_info(const type_info& rhs) = delete; // cannot be copied
    virtual ~type_info();
    size_t hash_code() const;
    _CRTIMP_PURE bool operator==(const type_info& rhs) const;
    type_info& operator=(const type_info& rhs) = delete; // cannot be copied
    _CRTIMP_PURE bool operator!=(const type_info& rhs) const;
    _CRTIMP_PURE int before(const type_info& rhs) const;
    size_t hash_code() const noexcept;
    _CRTIMP_PURE const char* name() const;
    _CRTIMP_PURE const char* raw_name() const;
};
```

Você não pode instanciar objetos da classe `type_info` diretamente, porque a classe tem apenas um construtor de cópia privada. A única maneira de construir um objeto de `type_info` (temporário) é usar o operador `typeid`. Como o operador de atribuição também é privado, não é possível copiar ou atribuir objetos da classe `type_info`.

`type_info::hash_code` define uma função de hash adequada para mapear valores do tipo `TypeInfo` para uma distribuição de valores de índice.

Os operadores `==` e `!=` podem ser usados para comparar a igualdade e desigualdade com outros objetos de `type_info`, respectivamente.

Não há nenhum link entre a ordem de agrupamento de tipos e relações de herança. Use a função de membro `type_info::before` para determinar a sequência de agrupamento dos tipos. Não há nenhuma garantia de que `type_info::before` produzirá o mesmo resultado em programas diferentes ou até mesmo em diferentes execuções do mesmo programa. Dessa maneira, `type_info::before` é semelhante ao operador de endereço de `(&)`.

A função membro `type_info::name` retorna uma `const char*` a uma cadeia de caracteres terminada em nulo que representa o nome legível do tipo. A memória apontada é armazenada em cache e nunca deve ser desalocada diretamente.

A função membro `type_info::raw_name` retorna um `const char*` a uma cadeia de caracteres terminada em nulo que representa o nome decorado do tipo de objeto. O nome é armazenado em sua forma decorada para economizar espaço. Consequentemente, essa função é mais rápida do que `type_info::name` porque não precisa desdecorar o nome. A cadeia de caracteres retornada pela função `type_info::raw_name` é útil em operações de comparação, mas não é legível. Se você precisar de uma cadeia de caracteres legível, use a função `type_info::name` em vez disso.

As informações de tipo serão geradas para classes polimórficas somente se a opção de compilador [/gr \(habilitar informações de tipo de tempo de execução\)](#) for especificada.

Confira também

[Informações de tipo em tempo de execução](#)

Instruções (C++)

15/04/2020 • 2 minutes to read • [Edit Online](#)

As instruções em C++ são os elementos do programa que controlam como e em que ordem os objetos são manipulados. Esta seção inclui:

- [Visão geral](#)
- [Declarações rotuladas](#)
- Categorias de instruções
 - [Declarações de expressão](#). Estas instruções avaliam uma expressão para verificar seus efeitos colaterais ou seu valor de retorno.
 - [Declarações nulas](#). Estas instruções podem ser fornecidas nos lugares em que uma instrução é exigida pela sintaxe do C++, mas nenhuma ação deve ser tomada.
 - [Declarações compostas](#). Estas instruções são grupos de instruções entre chaves ({}). Podem ser usadas onde quer que uma instrução simples possa ser usada.
 - [Declarações de seleção](#). Estas instruções realizam um teste, depois executam uma seção de código se o teste é avaliado como true (diferente de zero). Podem executar outra seção de código se o teste é avaliado como false.
 - [Declarações de iteração](#). Estas instruções promovem a execução repetida de um bloco de código até que um critério de término especificado seja encontrado.
 - [Declarações de salto](#). Estas instruções transferem o controle imediatamente para outro local da função ou retornam o controle à função.
 - [Declarações de declaração](#). As declarações introduzem um nome em um programa.

Para obter informações sobre as demonstrações de manipulação de exceções, consulte [O Tratamento de Exceções](#).

Confira também

[Referência de linguagem C++](#)

Visão geral de instruções C++

02/09/2020 • 2 minutes to read • [Edit Online](#)

As instruções C++ são executadas sequencialmente, exceto quando uma instrução de expressão, uma instrução de seleção, uma instrução de iteração ou uma instrução de salto modificam especificamente essa sequência.

As instruções podem ter os seguintes tipos:

Labeled-statement
expression-statement
compound-statement
selection-statement
iteration-statement
jump-statement
declaration-statement
try-throw-catch

Na maioria dos casos, a sintaxe da instrução C++ é idêntica à do ANSI C89. A principal diferença entre os dois é que, em C89, as declarações são permitidas apenas no início de um bloco; O C++ adiciona o *declaration-statement*, que efetivamente remove essa restrição. Isso permite que você apresente variáveis em um ponto no programa onde um valor de inicialização pré-computado pode ser calculado.

Declarar variáveis dentro de blocos também permite que você controle com precisão o escopo e o tempo de vida das variáveis.

Os artigos sobre instruções descrevem as seguintes palavras-chave do C++:

`break`
`case`
`catch`
`continue`
`default`
`do`

`else`
`__except`
`__finally`
`for`
`goto`

`if`
`__if_exists`
`__if_not_exists`
`__leave`
`return`

`switch`
`throw`
`__try`
`try`

while

Confira também

[Instruções](#)

Instruções rotuladas

02/09/2020 • 4 minutes to read • [Edit Online](#)

Rótulos são usados para transferir o controle do programa diretamente para a instrução especificada.

```
identifier : statement
case constant-expression : statement
default : statement
```

O escopo de um rótulo é a função inteira na qual é declarado.

Comentários

Há três tipos de instruções rotuladas. Todos usam dois-pontos para separar qualquer tipo do rótulo da instrução. O uso de rótulos padrão e case são específicos das instruções case.

```
#include <iostream>
using namespace std;

void test_label(int x) {

    if (x == 1){
        goto label1;
    }
    goto label2;

label1:
    cout << "in label1" << endl;
    return;

label2:
    cout << "in label2" << endl;
    return;
}

int main() {
    test_label(1); // in label1
    test_label(2); // in label2
}
```

A instrução goto

A aparência de um rótulo de *identificador* no programa de origem declara um rótulo. Somente uma instrução [goto](#) pode transferir o controle para um rótulo de *identificador*. O fragmento de código a seguir ilustra o uso da [goto](#) instrução e um rótulo de *identificador*:

Um rótulo não pode aparecer sozinho: deve estar sempre anexado a uma instrução. Se for necessário usar um rótulo sozinho, coloque uma instrução nula depois do rótulo.

O rótulo tem o escopo da função e não pode ser redeclarado dentro da função. No entanto, o mesmo nome pode ser usado como um rótulo em funções diferentes.

```

// labels_with_goto.cpp
// compile with: /EHsc
#include <iostream>
int main() {
    using namespace std;
    goto Test2;

    cout << "testing" << endl;

    Test2:
    cerr << "At Test2 label." << endl;
}

//Output: At Test2 label.

```

A instrução Case

Os rótulos que aparecem após a `case` palavra-chave também não podem aparecer fora de uma `switch` instrução. (Essa restrição também se aplica à `default` palavra-chave.) O fragmento de código a seguir mostra o uso correto de `case` Rótulos:

```

// Sample Microsoft Windows message processing loop.
switch( msg )
{
    case WM_TIMER:      // Process timer event.
        SetClassWord( hWnd, GCW_HICON, ahIcon[nIcon++]);
        ShowWindow( hWnd, SW_SHOWNA );
        nIcon %= 14;
        Yield();
        break;

    case WM_PAINT:
        memset( &ps, 0x00, sizeof(PAINTSTRUCT) );
        hDC = BeginPaint( hWnd, &ps );
        EndPaint( hWnd, &ps );
        break;

    default:
        // This choice is taken for all messages not specifically
        // covered by a case statement.

        return DefWindowProc( hWnd, Message, wParam, lParam );
        break;
}

```

Rótulos na instrução Case

Os rótulos que aparecem após a `case` palavra-chave também não podem aparecer fora de uma `switch` instrução. (Essa restrição também se aplica à `default` palavra-chave.) O fragmento de código a seguir mostra o uso correto de `case` Rótulos:

```

// Sample Microsoft Windows message processing loop.
switch( msg )
{
    case WM_TIMER:    // Process timer event.
        SetClassWord( hWnd, GCW_HICON, ahIcon[nIcon++] );
        ShowWindow( hWnd, SW_SHOWNA );
        nIcon %= 14;
        Yield();
        break;

    case WM_PAINT:
        // Obtain a handle to the device context.
        // BeginPaint will send WM_ERASEBKGND if appropriate.

        memset( &ps, 0x00, sizeof(PAINTSTRUCT) );
        hDC = BeginPaint( hWnd, &ps );

        // Inform Windows that painting is complete.

        EndPaint( hWnd, &ps );
        break;

    case WM_CLOSE:
        // Close this window and all child windows.

        KillTimer( hWnd, TIMER1 );
        DestroyWindow( hWnd );
        if ( hWnd == hWndMain )
            PostQuitMessage( 0 ); // Quit the application.
        break;

    default:
        // This choice is taken for all messages not specifically
        // covered by a case statement.

        return DefWindowProc( hWnd, Message, wParam, lParam );
        break;
}

```

Rótulos na instrução goto

A aparência de um rótulo de *identificador* no programa de origem declara um rótulo. Somente uma instrução **goto** pode transferir o controle para um rótulo de *identificador*. O fragmento de código a seguir ilustra o uso da **goto** instrução e um rótulo de *identificador*:

Um rótulo não pode aparecer sozinho: deve estar sempre anexado a uma instrução. Se for necessário usar um rótulo sozinho, coloque uma instrução nula depois do rótulo.

O rótulo tem o escopo da função e não pode ser redeclarado dentro da função. No entanto, o mesmo nome pode ser usado como um rótulo em funções diferentes.

```
// labels_with_goto.cpp
// compile with: /EHsc
#include <iostream>
int main() {
    using namespace std;
    goto Test2;

    cout << "testing" << endl;

    Test2:
    cerr << "At Test2 label." << endl;
// At Test2 label.
}
```

Confira também

[Visão geral das instruções do C++](#)

[Instrução switch \(C++\)](#)

Instrução de expressão

25/03/2020 • 2 minutes to read • [Edit Online](#)

Instruções de expressão fazem com as expressões sejam avaliadas. Nenhuma transferência de controle ou iteração ocorre como resultado de uma instrução de expressão.

A sintaxe da instrução de expressão é simplesmente

Sintaxe

```
[expression] ;
```

Comentários

Todas as expressões em uma instrução de expressão são avaliadas e todos os efeitos colaterais são concluídos antes que a próxima instrução seja executada. As instruções de expressão mais comuns são atribuições e chamadas de função. Como a expressão é opcional, um ponto-e-vírgula sozinho é considerado uma instrução de expressão vazia, conhecida como instrução [NULL](#) .

Confira também

[Visão geral das instruções C++](#)

Instrução nula

25/03/2020 • 2 minutes to read • [Edit Online](#)

A "instrução NULL" é uma instrução de expressão com a *expressão* ausente. É útil quando a sintaxe da linguagem pede por uma instrução, mas nenhuma avaliação de expressão. Consiste em um ponto e vírgula.

Em geral, as instruções nulas são usadas como espaços reservados em instruções de iteração ou como instruções nas quais são colocados rótulos no final de instruções ou funções compostas.

O seguinte fragmento de código mostra como copiar uma cadeia de caracteres para outra e incorpora a instrução nula:

```
// null_statement.cpp
char *myStrCpy( char *Dest, const char *Source )
{
    char *DestStart = Dest;

    // Assign value pointed to by Source to
    // Dest until the end-of-string 0 is
    // encountered.
    while( *Dest++ = *Source++ )
        ;    // Null statement.

    return DestStart;
}

int main()
{
}
```

Confira também

[Instrução de expressão](#)

Instruções compostas (blocos)

02/09/2020 • 2 minutes to read • [Edit Online](#)

Uma instrução composta consiste em zero ou mais instruções entre chaves ({}). Uma instrução composta pode ser usada em qualquer lugar em que uma instrução é esperada. As instruções compostas são comumente chamadas de "blocos".

Sintaxe

```
{ [ statement-list ] }
```

Comentários

O exemplo a seguir usa uma instrução composta como parte da *instrução* da `if` instrução (consulte [a instrução If](#) para obter detalhes sobre a sintaxe):

```
if( Amount > 100 )
{
    cout << "Amount was too large to handle\n";
    Alert();
}
else
{
    Balance -= Amount;
}
```

NOTE

Como uma declaração é uma instrução, uma declaração pode ser uma das instruções na *lista* de instruções. Portanto, os nomes declarados em uma instrução composta, mas não declarados explicitamente como estáticos, têm escopo e vida útil locais (para objetos). Consulte [escopo](#) para obter detalhes sobre o tratamento de nomes com escopo local.

Confira também

[Visão geral das instruções do C++](#)

Instruções de seleção (C++)

25/03/2020 • 2 minutes to read • [Edit Online](#)

As C++ instruções de seleção, [se](#) e a [opção](#), fornecem um meio para executar condicionalmente seções de código.

As instruções [_if_exists](#) e [_if_not_exists](#) permitem que você inclua código condicionalmente dependendo da existência de um símbolo.

Consulte os tópicos individuais sobre a sintaxe para cada instrução.

Confira também

[Visão geral das instruções C++](#)

Instrução if-else (C++)

02/09/2020 • 4 minutes to read • [Edit Online](#)

Controla a ramificação condicional. As instruções em *If-Block* serão executadas somente se a *expressão If* for avaliada como um valor diferente de zero (ou `true`). Se o valor de *expression* for diferente de zero, *instrução1* e quaisquer outras instruções no bloco serão executadas e o *else-Block*, se presente, será ignorado. Se o valor de *expression* for zero, o *If-Block* será ignorado e o *else-Block*, se presente, será executado. Expressões que são avaliadas como não zero são

- `true`
- um ponteiro não nulo,
- qualquer valor aritmético diferente de zero ou
- um tipo de classe que define uma conversão não ambígua para um tipo aritmético, booliano ou de ponteiro.
(Para obter informações sobre conversões, consulte [conversões padrão](#).)

Sintaxe

```
if ( expression )
{
    statement1;
    ...
}
else // optional
{
    statement2;
    ...
}

// C++17 - Visual Studio 2017 version 15.3 and later:
if ( initialization; expression )
{
    statement1;
    ...
}
else // optional
{
    statement2;
    ...
}

// C++17 - Visual Studio 2017 version 15.3 and later:
if constexpr (expression)
{
    statement1;
    ...
}
else // optional
{
    statement2;
    ...
}
```

Exemplo

```

// if_else_statement.cpp
#include <iostream>

using namespace std;

class C
{
public:
    void do_something(){}
};

void init(C){}
bool is_true() { return true; }
int x = 10;

int main()
{
    if (is_true())
    {
        cout << "b is true!\n"; // executed
    }
    else
    {
        cout << "b is false!\n";
    }

    // no else statement
    if (x == 10)
    {
        x = 0;
    }

    C* c;
    init(c);
    if (c)
    {
        c->do_something();
    }
    else
    {
        cout << "c is null!\n";
    }
}

```

instrução If com um inicializador

Visual Studio 2017 versão 15,3 e posterior (disponível com [/std: c++ 17](#)): uma `if` instrução também pode conter uma expressão que declara e inicializa uma variável nomeada. Use essa forma da instrução If-quando a variável só for necessária dentro do escopo do bloco If.

Exemplo

```

#include <iostream>
#include <mutex>
#include <map>
#include <string>
#include <algorithm>

using namespace std;

map<int, string> m;
mutex mx;
bool shared_flag; // guarded by mx
void unsafe_operation() {}

int main()
{
    if (auto it = m.find(10); it != m.end())
    {
        cout << it->second;
        return 0;
    }

    if (char buf[10]; fgets(buf, 10, stdin))
    {
        m[0] += buf;
    }

    if (lock_guard<mutex> lock(mx); shared_flag)
    {
        unsafe_operation();
        shared_flag = false;
    }

    string s{ "if" };
    if (auto keywords = { "if", "for", "while" }; any_of(keywords.begin(), keywords.end(), [&s](const char* kw) { return s == kw; }))
    {
        cout << "Error! Token must not be a keyword\n";
    }
}

```

Em todos os formulários da `if` instrução, *expressão*, que pode ter qualquer valor, exceto uma estrutura, é avaliada, incluindo todos os efeitos colaterais. O controle passa da `if` instrução para a próxima instrução no programa, a menos que um dos s da *instrução* contenha um `Break`, `continue` ou `goto`.

A `else` cláusula de uma `if...else` instrução é associada à instrução anterior mais próxima `if` no mesmo escopo que não tem uma `else` instrução correspondente.

Visual Studio 2017 versão 15,3 e posterior (disponível com [/std: c++ 17](#)): em modelos de função, você pode usar uma instrução `If constexpr` para tomar decisões de ramificação em tempo de compilação sem a necessidade de recorrer a várias sobrecargas de função. Por exemplo, você pode escrever uma única função que manipula o desempacotamento de parâmetro (nenhuma sobrecarga de parâmetro zero é necessária):

```
template <class T, class... Rest>
void f(T&& t, Rest&&... r)
{
    // handle t
    do_something(t);

    // handle r conditionally
    if constexpr (sizeof...(r))
    {
        f(r...);
    }
    else
    {
        g(r...);
    }
}
```

Confira também

[Instruções de seleção](#)

[Palavras-chave](#)

[Instrução switch \(C++\)](#)

Instrução `__if_exists`

02/09/2020 • 2 minutes to read • [Edit Online](#)

A `__if_exists` instrução testa se o identificador especificado existe. Se o identificador especificado existir, o bloco de instrução especificado é executado.

Sintaxe

```
__if_exists ( identifier ) {
    statements
}
```

Parâmetros

ID

O identificador cuja existência você deseja testar.

instruções

Uma ou mais instruções para executar se o *identificador* existir.

Comentários

Caution

Para obter os resultados mais confiáveis, use a `__if_exists` instrução sob as restrições a seguir.

- Aplique a `__if_exists` instrução somente a tipos simples, não a modelos.
- Aplique a `__if_exists` instrução aos identificadores dentro ou fora de uma classe. Não aplique a `__if_exists` instrução a variáveis locais.
- Use a `__if_exists` instrução somente no corpo de uma função. Fora do corpo de uma função, a `__if_exists` instrução pode testar apenas tipos totalmente definidos.
- Quando você testa funções sobrecarregadas, não é possível testar um formato específico de sobrecarga.

O complemento à `__if_exists` instrução é a instrução [__if_not_exists](#).

Exemplo

Observe que este exemplo usa modelos, o que não é recomendável.

```

// the_if_exists_statement.cpp
// compile with: /EHsc
#include <iostream>

template<typename T>
class X : public T {
public:
    void Dump() {
        std::cout << "In X<T>::Dump()" << std::endl;

        __if_exists(T::Dump) {
            T::Dump();
        }

        __if_not_exists(T::Dump) {
            std::cout << "T::Dump does not exist" << std::endl;
        }
    }
};

class A {
public:
    void Dump() {
        std::cout << "In A::Dump()" << std::endl;
    }
};

class B {};

bool g_bFlag = true;

class C {
public:
    void f(int);
    void f(double);
};

int main() {
    X<A> x1;
    X<B> x2;

    x1.Dump();
    x2.Dump();

    __if_exists(::g_bFlag) {
        std::cout << "g_bFlag = " << g_bFlag << std::endl;
    }

    __if_exists(C::f) {
        std::cout << "C::f exists" << std::endl;
    }

    return 0;
}

```

Saída

```

In X<T>::Dump()
In A::Dump()
In X<T>::Dump()
T::Dump does not exist
g_bFlag = 1
C::f exists

```

Confira também

[Instruções de seleção](#)

[Palavras-chave](#)

[Instrução __if_not_exists](#)

Instrução __if_not_exists

02/09/2020 • 2 minutes to read • [Edit Online](#)

A `__if_not_exists` instrução testa se o identificador especificado existe. Se o identificador especificado não existir, o bloco de instrução especificado é executado.

Sintaxe

```
__if_not_exists ( identifier ) {  
    statements  
};
```

Parâmetros

ID

O identificador cuja existência você deseja testar.

instruções

Uma ou mais instruções a serem executadas se o *identificador* não existir.

Comentários

Caution

Para obter os resultados mais confiáveis, use a `__if_not_exists` instrução sob as restrições a seguir.

- Aplique a `__if_not_exists` instrução somente a tipos simples, não a modelos.
- Aplique a `__if_not_exists` instrução aos identificadores dentro ou fora de uma classe. Não aplique a `__if_not_exists` instrução a variáveis locais.
- Use a `__if_not_exists` instrução somente no corpo de uma função. Fora do corpo de uma função, a `__if_not_exists` instrução pode testar apenas tipos totalmente definidos.
- Quando você testa funções sobrecarregadas, não é possível testar um formato específico de sobrecarga.

O complemento à `__if_not_exists` instrução é a instrução [__if_exists](#).

Exemplo

Para obter um exemplo sobre como usar o `__if_not_exists`, consulte [__if_exists instrução](#).

Confira também

[Instruções de seleção](#)

[Palavras-chave](#)

[Instrução __if_exists](#)

:::no-loc(switch)::: instrução (C++)

02/09/2020 • 8 minutes to read • [Edit Online](#)

Permite a seleção entre várias seções de código, dependendo do valor de uma expressão integral.

Sintaxe

```
selection-statement :  
  ::::no-loc(switch)::: ( init-statement ::::no-loc(opt)::: C++ 17 condition ) statement
```

```
init-statement :  
  expression-statement  
  simple-declaration
```

```
condition :  
  expression  
  attribute-specifier-seq ::::no-loc(opt)::: decl-specifier-seq declarator brace-or-equal-initializer
```

```
Labeled-statement :  
  ::::no-loc(case)::: constant-expression : statement  
  ::::no-loc(default)::: : statement
```

Comentários

Uma ::::no-loc(switch)::: instrução faz com que o controle seja transferido para um Labeled-statement em seu corpo de instrução, dependendo do valor de condition .

O condition deve ter um tipo integral ou ser um tipo de classe que tenha uma conversão não ambígua para o tipo integral. A promoção integral ocorre conforme descrito em [conversões padrão](#).

O ::::no-loc(switch)::: corpo da instrução consiste em uma série de ::::no-loc(case)::: rótulos e um ::::no-loc(opt)::: rótulo de ional ::::no-loc(default)::: . Um Labeled-statement é um desses rótulos e as instruções a seguir. As instruções rotuladas não são requisitos sintáticos, mas a ::::no-loc(switch)::: instrução não faz sentido sem elas. Dois constant-expression valores em ::::no-loc(case)::: instruções podem ser avaliados com o mesmo valor. O ::::no-loc(default)::: rótulo pode aparecer apenas uma vez. A ::::no-loc(default)::: instrução geralmente é colocada no final, mas pode aparecer em qualquer lugar no ::::no-loc(switch)::: corpo da instrução. Um ::::no-loc(case)::: ::::no-loc(default)::: rótulo ou só pode aparecer dentro de uma ::::no-loc(switch)::: instrução.

O constant-expression em cada ::::no-loc(case)::: rótulo é convertido em um valor constante que é do mesmo tipo que condition . Em seguida, ele é comparado com a condition igualdade. O controle passa para a primeira instrução após o ::::no-loc(case)::: constant-expression valor que corresponde ao valor de condition . O comportamento resultante é mostrado na tabela a seguir.

::::no-loc(switch)::: comportamento da instrução

CONDICÃO	AÇÃO
O valor convertido corresponde ao da expressão de controle promovida.	O controle é transferido para a instrução após esse rótulo.
Nenhuma das constantes corresponde às constantes nos <code>::::no-loc(case):::</code> Rótulos; um <code>::::no-loc(default):::</code> rótulo está presente.	O controle é transferido para o <code>::::no-loc(default):::</code> rótulo.
Nenhuma das constantes corresponde às constantes nos <code>::::no-loc(case):::</code> Rótulos; nenhum <code>::::no-loc(default):::</code> rótulo está presente.	O controle é transferido para a instrução após a <code>::::no-loc(switch):::</code> instrução.

Se uma expressão correspondente for encontrada, a execução poderá continuar mais tarde `::::no-loc(case):::` ou `::::no-loc(default):::` Rótulos. A `::::no-loc(break):::` instrução é usada para interromper a execução e transferir o controle para a instrução após a `::::no-loc(switch):::` instrução. Sem uma `::::no-loc(break):::` instrução, todas as instruções do `::::no-loc(case):::` rótulo correspondente ao final do `::::no-loc(switch):::`, incluindo o `::::no-loc(default):::`, são executadas. Por exemplo:

```
// ::::no-loc(switch):::_statement1.cpp
#include <stdio.h>

int main() {
    const char *buffer = "Any character stream";
    int upper:::no-loc(case):::_A, lower:::no-loc(case):::_a, other;
    char c;
    upper:::no-loc(case):::_A = lower:::no-loc(case):::_a = other = 0;

    ::::no-loc(while)::: ( c = *buffer++ ) // Walks buffer until NULL
    {
        ::::no-loc(switch)::: ( c )
        {
            ::::no-loc(case)::: 'A':
                upper:::no-loc(case):::_A++;
            ::::no-loc(break):::;
            ::::no-loc(case)::: 'a':
                lower:::no-loc(case):::_a++;
            ::::no-loc(break):::;
            ::::no-loc(default):::
                other++;
        }
    }
    printf_s( "\nUpper:::no-loc(case)::: A: %d\nLower:::no-loc(case)::: a: %d\nTotal: %d\n",
        upper:::no-loc(case):::_A, lower:::no-loc(case):::_a, (upper:::no-loc(case):::_A + lower:::no-loc(case):::_a + other) );
}
```

No exemplo acima, `upper:::no-loc(case):::_A` será incrementado se `c` for um Upper `:::no-loc(case)::: 'A'`. A `::::no-loc(break):::` instrução depois `upper:::no-loc(case):::_A++` encerra a execução do `::::no-loc(switch):::` corpo da instrução e o controle passa para o `::::no-loc(while):::` loop. Sem a `::::no-loc(break):::` instrução, a execução "passa" para a próxima instrução rotulada, de modo que `lower:::no-loc(case):::_a` e `other` também seria incrementada. Uma finalidade semelhante é servida pela `::::no-loc(break):::` instrução para `::::no-loc(case)::: 'a'`. Se `c` for um menor `:::no-loc(case)::: 'a'`, `lower:::no-loc(case):::_a` será incrementado e a `::::no-loc(break):::` instrução terminará o `::::no-loc(switch):::` corpo da instrução. Se `c` não for um `'a'` ou `'A'`, a `::::no-loc(default):::` instrução será executada.

Visual Studio 2017 e posterior: (disponível com [/std: c++ 17](#)) o `[[fallthrough]]` atributo é especificado no

padrão C++ 17. Você pode usá-lo em uma `:::no-loc(switch):::` instrução. É uma dica para o compilador, ou qualquer pessoa que lê o código, o comportamento de passagem é intencional. O compilador do Microsoft C++ atualmente não avisa sobre o comportamento do fallthrough, portanto, esse atributo não tem nenhum efeito sobre o comportamento do compilador. No exemplo, o atributo é aplicado a uma instrução vazia dentro da instrução rotulada não finalizada. Em outras palavras, o ponto e vírgula é necessário.

```
int main()
{
    int n = 5;
    ::::no-loc(switch)::: (n
    {

        ::::no-loc(case)::: 1:
        a();
        ::::no-loc(break):::;
        ::::no-loc(case)::: 2:
        b();
        d();
        [[fallthrough]]; // I meant to do this!
        ::::no-loc(case)::: 3:
        c();
        ::::no-loc(break):::;
        ::::no-loc(default):::
        d();
        ::::no-loc(break):::;
    }

    return 0;
}
```

Visual Studio 2017 versão 15,3 e posterior (disponível com [/std:c++ 17](#)). Uma `:::no-loc(switch):::` instrução pode ter uma `init-statement` cláusula, que termina com um ponto e vírgula. Ele apresenta e Inicializa uma variável cujo escopo é limitado ao bloco da `:::no-loc(switch):::` instrução:

```
::::no-loc(switch)::: (Gadget gadget(args); auto s = gadget.get_status())
{
    ::::no-loc(case)::: status::good:
    gadget.zip();
    ::::no-loc(break):::;
    ::::no-loc(case)::: status::bad:
    throw BadGadget();
};
```

Um bloco interno de uma `:::no-loc(switch):::` instrução pode conter definições com inicializadores, desde que eles estejam *acessíveis*, ou seja, não sejam ignorados por todos os caminhos de execução possíveis. Os nomes introduzidos por meio dessas declarações têm escopo local. Por exemplo:

```

// ::::no-loc(switch):::_statement2.cpp
// C2360 expected
#include <iostream>
using namespace std;
int main(int argc, char *argv[])
{
    ::::no-loc(switch):::( tolower( *argv[1] ) )
    {
        // Error. Unreachable declaration.
        char szChEntered[] = "Character entered was: ";

        ::::no-loc(case)::: 'a' :
        {
            // Declaration of szChEntered OK. Local scope.
            char szChEntered[] = "Character entered was: ";
            cout << szChEntered << "a\n";
        }
        ::::no-loc(break):::;

        ::::no-loc(case)::: 'b' :
        {
            // Value of szChEntered undefined.
            cout << szChEntered << "b\n";
        }
        ::::no-loc(break):::;

        ::::no-loc(default):::
        {
            // Value of szChEntered undefined.
            cout << szChEntered << "neither a nor b\n";
        }
        ::::no-loc(break):::;
    }
}

```

Uma `::::no-loc(switch):::` instrução pode ser aninhada. Quando aninhados, `::::no-loc(case):::` os `::::no-loc(default):::` rótulos ou se associam à `::::no-loc(switch):::` instrução mais próxima que os coloca.

Comportamento específico da Microsoft

O Microsoft C++ não limita o número de `::::no-loc(case):::` valores em uma `::::no-loc(switch):::` instrução. O número é limitado somente pela memória disponível.

Confira também

[Instruções de seleção](#)

[Palavras-chave](#)

Instruções de iteração (C++)

02/09/2020 • 2 minutes to read • [Edit Online](#)

As instruções de iteração fazem com que as instruções (ou instruções compostas) sejam executadas nenhuma ou mais vezes, de acordo com os critérios de término de loop. Quando essas instruções são instruções compostas, elas são executadas em ordem, exceto quando a instrução [Break](#) ou a instrução [continue](#) é encontrada.

O C++ fornece quatro instruções de iteração – [enquanto](#), [do](#), [para](#) e o [baseado em intervalo](#). Cada uma dessas iterações até que sua expressão de término seja avaliada como zero (false), ou até que o encerramento do loop seja forçado com uma [break](#) instrução. A tabela a seguir resume essas instruções e suas ações; cada uma delas é discutida em detalhes nas seções seguintes.

Instruções de iteração

DE	AVALIADA COMO	INICIALIZAÇÃO	INCREMENTO
<code>while</code>	Topo do loop	Não	Não
<code>do</code>	Final do loop	Não	Não
<code>for</code>	Topo do loop	Sim	Sim
<code>range-based for</code>	Topo do loop	Sim	Sim

A parte da instrução de uma instrução de iteração não pode ser uma declaração. No entanto, pode ser uma instrução composta que contenha uma declaração.

Confira também

[Visão geral das instruções do C++](#)

Instrução while (C++)

02/09/2020 • 2 minutes to read • [Edit Online](#)

Executa a *instrução* repetidamente até que a *expressão* seja avaliada como zero.

Sintaxe

```
while ( expression )
    statement
```

Comentários

O teste de *expressão* ocorre antes de cada execução do loop; Portanto, um `while` loop é executado zero ou mais vezes. A *expressão* deve ser de um tipo integral, um tipo de ponteiro ou um tipo de classe com uma conversão não ambígua para um tipo integral ou de ponteiro.

Um `while` loop também pode terminar quando um [Break](#), [goto](#) ou [Return](#) dentro do corpo da instrução `for` executado. Use [continuar](#) para encerrar a iteração atual sem sair do `while` loop. `continue` passa o controle para a próxima iteração do `while` loop.

O código a seguir usa um `while` loop para cortar sublinhados à direita de uma cadeia de caracteres:

```
// while_statement.cpp

#include <string.h>
#include <stdio.h>
char *trim( char *szSource )
{
    char *pszEOS = 0;

    // Set pointer to character before terminating NULL
    pszEOS = szSource + strlen( szSource ) - 1;

    // iterate backwards until non '_' is found
    while( (pszEOS >= szSource) && (*pszEOS == '_') )
        *pszEOS-- = '\0';

    return szSource;
}
int main()
{
    char szbuf[] = "12345____";

    printf_s("\nBefore trim: %s", szbuf);
    printf_s("\nAfter trim: %s\n", trim(szbuf));
}
```

A condição de término é avaliada na parte superior do loop. Se não houver um sublinhado à direita, o loop nunca será executado.

Confira também

[Instruções de iteração](#)

[Palavras-chave](#)

[Instrução do-while \(C++\)](#)

[Instrução for \(C++\)](#)

[Instrução for com base em intervalo \(C++\)](#)

Instrução do-while (C++)

25/03/2020 • 2 minutes to read • [Edit Online](#)

Executa uma *instrução* repetidamente até que a condição de término especificada (a *expressão*) seja avaliada como zero.

Sintaxe

```
do
    statement
while ( expression ) ;
```

Comentários

O teste da condição de encerramento é feito após cada execução do loop; Portanto, um loop **do-while** executa uma ou mais vezes, dependendo do valor da expressão de encerramento. A instrução **do-while** também pode terminar quando uma instrução [break](#), [goto](#) ou [return](#) é executada no corpo da instrução.

A *expressão* deve ter o tipo aritmético ou ponteiro. A execução procede da seguinte maneira:

1. O corpo da instrução é executado.
2. Em seguida, a *expressão* é avaliada. Se a *expressão* for falsa, a instrução **do-while** será finalizada e o controle será passado para a próxima instrução no programa. Se a *expressão* for verdadeira (diferente de zero), o processo será repetido, começando da etapa 1.

Exemplo

O exemplo a seguir demonstra a instrução **do-while** :

```
// do_while_statement.cpp
#include <stdio.h>
int main()
{
    int i = 0;
    do
    {
        printf_s("\n%d",i++);
    } while (i < 3);
}
```

Confira também

[Instruções de iteração](#)

[Palavras-chave](#)

[Instrução while \(C++\)](#)

[Instrução for \(C++\)](#)

[Instrução for baseada intervalo \(C++\)](#)

for instrução (C++)

02/09/2020 • 6 minutes to read • [Edit Online](#)

Executa uma instrução repetidamente até que a condição se torne falsa. Para obter informações sobre a instrução baseada em intervalo `for`, consulte [demonstrativo baseado em intervalo `for` \(C++\)](#).

Sintaxe

```
for ( init-expression ; cond-expression ; Loop-expression )  
    statement
```

Comentários

Use a `for` instrução para construir loops que devem executar um número de vezes especificado.

A `for` instrução consiste em três partes opcionais, conforme mostrado na tabela a seguir.

elementos de loop for

NOME DA SINTAXE	QUANDO EXECUTADO	DESCRIÇÃO
<code>init-expression</code>	Antes de qualquer outro elemento da <code>for</code> instrução, <code>init-expression</code> é executado apenas uma vez. Em seguida, o controle passa para <code>cond-expression</code> .	Muitas vezes usado para inicializar índices de loop. Ele pode conter expressões ou declarações.
<code>cond-expression</code>	Antes da execução de cada iteração de <code>statement</code> , incluindo a primeira iteração. <code>statement</code> será executado somente se <code>cond-expression</code> for avaliado como true (diferente de zero).	Uma expressão que é avaliada para um tipo integral ou um tipo de classe que tem uma conversão ambígua para um tipo integral. Geralmente usado para testar critérios de encerramento de loop.
<code>Loop-expression</code>	No final de cada iteração de <code>statement</code> . Após <code>Loop-expression</code> ser executado, <code>cond-expression</code> é avaliado.	Geralmente usado para incrementar índices de loop.

Os exemplos a seguir mostram diferentes maneiras de usar a `for` instrução.

```
#include <iostream>
using namespace std;

int main() {
    // The counter variable can be declared in the init-expression.
    for (int i = 0; i < 2; i++ ){
        cout << i;
    }
    // Output: 01
    // The counter variable can be declared outside the for loop.
    int i;
    for (i = 0; i < 2; i++){
        cout << i;
    }
    // Output: 01
    // These for loops are the equivalent of a while loop.
    i = 0;
    while (i < 2){
        cout << i++;
    }
    // Output: 01
}
```

`init-expression` e `Loop-expression` pode conter várias instruções separadas por vírgulas. Por exemplo:

```
#include <iostream>
using namespace std;

int main(){
    int i, j;
    for ( i = 5, j = 10 ; i + j < 20; i++, j++ ) {
        cout << "i + j = " << (i + j) << '\n';
    }
}
// Output:
i + j = 15
i + j = 17
i + j = 19
```

`Loop-expression` pode ser incrementado ou diminuído ou modificado de outras maneiras.

```
#include <iostream>
using namespace std;

int main(){
for (int i = 10; i > 0; i--) {
    cout << i << ' ';
}
// Output: 10 9 8 7 6 5 4 3 2 1
for (int i = 10; i < 20; i = i+2) {
    cout << i << ' ';
}
// Output: 10 12 14 16 18
```

Um `for` loop termina quando um `break`, `Returnou` `goto` (para uma instrução rotulada fora do `for` loop) dentro `statement` é executado. Uma `continue` instrução em um `for` loop encerra apenas a iteração atual.

Se `cond-expression` for omitido, ele será considerado `true` e o `for` loop não terminará sem um `break`, `return` ou `goto` dentro de `statement`.

Embora os três campos da `for` instrução sejam normalmente usados para inicialização, teste para

encerramento e incrementos, eles não estão restritos a esses usos. Por exemplo, o código a seguir imprime os número de 0 a 4. Nesse caso, `statement` é a instrução NULL:

```
#include <iostream>
using namespace std;

int main()
{
    int i;
    for( i = 0; i < 5; cout << i << '\n', i++){
        ;
    }
}
```

for loops e o C++ padrão

O padrão C++ diz que uma variável declarada em um `for` loop deve sair do escopo depois que o `for` loop termina. Por exemplo:

```
for (int i = 0 ; i < 5 ; i++) {
    // do something
}
// i is now out of scope under /Za or /Zc:forScope
```

Por padrão, em `/ze`, uma variável declarada em um `for` loop permanece no escopo até que o `for` escopo de delimitação do loop seja encerrado.

`/Zc:forScope` habilita o comportamento padrão de variáveis declaradas em loops for sem a necessidade de especificar `/za`.

Também é possível usar as diferenças de escopo do `for` loop para redeclarar as variáveis abaixo da seguinte `/ze` maneira:

```
// for_statement5.cpp
int main(){
    int i = 0; // hidden by var with same name declared in for loop
    for ( int i = 0 ; i < 3; i++ ) {}

    for ( int i = 0 ; i < 3; i++ ) {}
}
```

Esse comportamento imita mais de forma semelhante o comportamento padrão de uma variável declarada em um `for` loop, que requer variáveis declaradas em um `for` loop para sair do escopo após a conclusão do loop. Quando uma variável é declarada em um `for` loop, o compilador a promove internamente a uma variável local no `for` escopo de delimitação do loop. Ele é promovido mesmo que já exista uma variável local com o mesmo nome.

Consulte também

[Instruções de iteração](#)

[Palavras-chave](#)

[instrução while \(C++\)](#)

[instrução do-while \(C++\)](#)

[Instrução for com base em intervalo \(C++\)](#)

Instrução for com base em intervalo (C++)

02/09/2020 • 4 minutes to read • [Edit Online](#)

Executa `statement` repetidamente e em sequência para cada elemento em `expression`.

Sintaxe

```
for ( declaração de intervalo para : expressão de** )
    instrução
```

Comentários

Use a instrução baseada em intervalo `for` para construir loops que devem ser executados por meio de um *intervalo*, que é definido como qualquer coisa na qual você possa iterar — por exemplo, `std::vector` ou qualquer outra sequência de biblioteca padrão do C++ cujo intervalo seja definido por um `begin()` e `end()`. O nome declarado na `for-range-declaration` parte é local para a `for` instrução e não pode ser declarado novamente em `expression` ou `statement`. Observe que a `auto` palavra-chave é preferida na `for-range-declaration` parte da instrução.

Novidade no Visual Studio 2017: `for` Os loops baseados em intervalo não exigem mais que `begin()` e `end()` retornem objetos do mesmo tipo. Isso permite que o `end()` retorne um objeto Sentinel, como usado por intervalos, conforme definido na proposta intervalos-v3. Para obter mais informações, consulte [generalizando o For loop baseado em intervalo](#) e a [biblioteca Range-v3 no GitHub](#).

Este código mostra como usar loops baseados em intervalo `for` para iterar por meio de uma matriz e um vetor:

```

// range-based-for.cpp
// compile by using: cl /EHsc /nologo /W4
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    // Basic 10-element integer array.
    int x[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

    // Range-based for loop to iterate through the array.
    for( int y : x ) { // Access by value using a copy declared as a specific type.
        // Not preferred.
        cout << y << " ";
    }
    cout << endl;

    // The auto keyword causes type inference to be used. Preferred.

    for( auto y : x ) { // Copy of 'x', almost always undesirable
        cout << y << " ";
    }
    cout << endl;

    for( auto &y : x ) { // Type inference by reference.
        // Observes and/or modifies in-place. Preferred when modify is needed.
        cout << y << " ";
    }
    cout << endl;

    for( const auto &y : x ) { // Type inference by const reference.
        // Observes in-place. Preferred when no modify is needed.
        cout << y << " ";
    }
    cout << endl;
    cout << "end of integer array test" << endl;
    cout << endl;

    // Create a vector object that contains 10 elements.
    vector<double> v;
    for (int i = 0; i < 10; ++i) {
        v.push_back(i + 0.14159);
    }

    // Range-based for loop to iterate through the vector, observing in-place.
    for( const auto &j : v ) {
        cout << j << " ";
    }
    cout << endl;
    cout << "end of vector test" << endl;
}

```

Esta é a saída:

```

1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
end of integer array test

0.14159 1.14159 2.14159 3.14159 4.14159 5.14159 6.14159 7.14159 8.14159 9.14159
end of vector test

```

Um loop baseado em intervalo `for` termina quando um desses em `statement` é executado: a `break` , `return` ou `goto` a uma instrução rotulada fora do loop baseado em intervalo `for` . Uma `continue` instrução em um loop baseado em intervalo `for` encerra apenas a iteração atual.

Tenha em mente estes fatos sobre o baseado em intervalo `for` :

- Reconhece matrizes automaticamente.
- Reconhece os contêineres que têm `.begin()` e `.end()` .
- Usa a pesquisa dependente de argumentos `begin()` e `end()` para qualquer outra coisa.

Confira também

`auto`

[Instruções de iteração](#)

[Palavras-chave](#)

`while` Instrução (C++)

`do-while` Instrução (C++)

`for` Instrução (C++)

Instruções de salto (C++)

25/03/2020 • 2 minutes to read • [Edit Online](#)

Uma instrução de salto do C++ executa uma transferência de controle local imediata.

Sintaxe

```
break;  
continue;  
return [expression];  
goto identifier;
```

Comentários

Consulte os seguintes tópicos para obter uma descrição das instruções de salto do C++:

- [Instrução Break](#)
- [Instrução continue](#)
- [Instrução return](#)
- [Instrução goto](#)

Confira também

[Visão geral das instruções C++](#)

Instrução break (C++)

02/09/2020 • 3 minutes to read • [Edit Online](#)

A `break` instrução encerra a execução do loop delimitador ou da instrução condicional mais próxima na qual ele aparece. O controle passa para a instrução que segue o encerramento da instrução, se houver.

Sintaxe

```
break;
```

Comentários

A `break` instrução é usada com a instrução `switch` condicional e com as `do` instruções do `for` loop do `as`, e `while`.

Em uma `switch` instrução, a `break` instrução faz com que o programa execute a próxima instrução fora da `switch` instrução. Sem uma `break` instrução, cada instrução do rótulo correspondente `case` ao final da `switch` instrução, incluindo a `default` cláusula, é executada.

Em loops, a `break` instrução encerra a execução da `do` instrução, ou, delimitadora mais próxima `for` `while`. O controle passa para a instrução que segue a instrução encerrada, se houver.

Em instruções aninhadas, a `break` instrução encerra apenas a `do` instrução, `for` `switch` ou `while` que a inclui imediatamente. Você pode usar uma `return` `goto` instrução or para transferir o controle de estruturas mais profundamente aninhadas.

Exemplo

O código a seguir mostra como usar a `break` instrução em um `for` loop.

```

#include <iostream>
using namespace std;

int main()
{
    // An example of a standard for loop
    for (int i = 1; i < 10; i++)
    {
        if (i == 4) {
            break;
        }
        cout << i << '\n';
    }

    // An example of a range-based for loop
    int nums []{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    for (int i : nums) {
        if (i == 4) {
            break;
        }
        cout << i << '\n';
    }
}

```

In each case:

1
2
3

O código a seguir mostra como usar `break` o em um `while` loop e um `do` loop.

```

#include <iostream>
using namespace std;

int main() {
    int i = 0;

    while (i < 10) {
        if (i == 4) {
            break;
        }
        cout << i << '\n';
        i++;
    }

    i = 0;
    do {
        if (i == 4) {
            break;
        }
        cout << i << '\n';
        i++;
    } while (i < 10);
}

```

In each case:

0123

O código a seguir mostra como usar `break` em uma instrução switch. Você deve usar `break` em todos os casos se desejar manipular cada caso separadamente; se você não usar `break`, a execução do código passará

para o próximo caso.

```
#include <iostream>
using namespace std;

enum Suit{ Diamonds, Hearts, Clubs, Spades };

int main() {

    Suit hand;
    . . .
    // Assume that some enum value is set for hand
    // In this example, each case is handled separately
    switch (hand)
    {
        case Diamonds:
            cout << "got Diamonds \n";
            break;
        case Hearts:
            cout << "got Hearts \n";
            break;
        case Clubs:
            cout << "got Clubs \n";
            break;
        case Spades:
            cout << "got Spades \n";
            break;
        default:
            cout << "didn't get card \n";
    }
    // In this example, Diamonds and Hearts are handled one way, and
    // Clubs, Spades, and the default value are handled another way
    switch (hand)
    {
        case Diamonds:
        case Hearts:
            cout << "got a red card \n";
            break;
        case Clubs:
        case Spades:
        default:
            cout << "didn't get a red card \n";
    }
}
```

Confira também

[Instruções de atalho](#)

[Palavras-chave](#)

[Instrução Continue](#)

Instrução continue (C++)

02/09/2020 • 2 minutes to read • [Edit Online](#)

Força a transferência de controle para a expressão de controle do **loop** do menor delimitador, **para**, ou **while**.

Sintaxe

```
continue;
```

Comentários

As instruções restantes na iteração atual não são executadas. A próxima iteração do loop é determinada da seguinte maneira:

- Em um `do` `while` loop ou, a próxima iteração começa reavaliando a expressão de controle `do` da `while` instrução or.
- Em um `for` loop (usando a sintaxe `for(<init-expr> ; <cond-expr> ; <loop-expr>)`), a `<loop-expr>` cláusula é executada. A cláusula `<cond-expr>` é reavaliada e, dependendo do resultado, o loop é encerrado ou ocorre outra iteração.

O exemplo a seguir mostra como a `continue` instrução pode ser usada para ignorar seções de código e começar a próxima iteração de um loop.

Exemplo

```
// continue_statement.cpp
#include <stdio.h>
int main()
{
    int i = 0;
    do
    {
        i++;
        printf_s("before the continue\n");
        continue;
        printf("after the continue, should never print\n");
    } while (i < 3);

    printf_s("after the do loop\n");
}
```

```
before the continue
before the continue
before the continue
after the do loop
```

Confira também

[Instruções de atalho](#)

[Palavras-chave](#)

Instrução return (C++)

02/09/2020 • 2 minutes to read • [Edit Online](#)

Finaliza a execução de uma função e retorna o controle para a função de chamada (ou para o sistema operacional, se o controle for transferido da função `main`). A execução é retomada na função de chamada no ponto imediatamente após a chamada.

Sintaxe

```
return [expression];
```

Comentários

A cláusula `expression`, caso exista, é convertida no tipo especificado na declaração da função, como se uma inicialização estivesse sendo executada. A conversão do tipo da expressão para o `return` tipo da função pode criar objetos temporários. Para obter mais informações sobre como e quando temporaries são criados, consulte [objetos temporários](#).

O valor da cláusula `expression` é retornado à função de chamada. Se a expressão for omitida, o valor de retorno da função será indefinido. Construtores e destruidores, e funções do tipo `void`, não podem especificar uma expressão na `return` instrução. As funções de todos os outros tipos devem especificar uma expressão na `return` instrução.

Quando o fluxo de controle sai do bloco que está delimitando a definição da função, o resultado é o mesmo que seria se uma `return` instrução sem uma expressão tivesse sido executada. Isso não é válido para funções que são declaradas como retornando um valor.

Uma função pode ter qualquer número de `return` instruções.

O exemplo a seguir usa uma expressão com uma `return` instrução para obter o maior de dois inteiros.

Exemplo

```
// return_statement2.cpp
#include <stdio.h>

int max ( int a, int b )
{
    return ( a > b ? a : b );
}

int main()
{
    int nOne = 5;
    int nTwo = 7;

    printf_s("\n%d is bigger\n", max( nOne, nTwo ) );
}
```

Confira também

[Instruções de atalho](#)

[Palavras-chave](#)

Instrução goto (C++)

02/09/2020 • 2 minutes to read • [Edit Online](#)

A `goto` instrução transfere incondicionalmente o controle para a instrução rotulada pelo identificador especificado.

Sintaxe

```
goto identifier;
```

Comentários

A instrução rotulada designada por `identifier` deve estar na função atual. Todos os nomes de `identifier` são membros de um namespace interno e, portanto, não interferem em outros identificadores.

Um rótulo de instrução é significativo apenas para uma `goto` instrução; caso contrário, os rótulos de instrução são ignorados. Os rótulos não podem ser redeclarados.

Uma `goto` instrução não tem permissão para transferir o controle para um local que ignora a inicialização de qualquer variável que esteja no escopo nesse local. O exemplo a seguir gera C2362:

```
int goto_fn(bool b)
{
    if (!b)
    {
        goto exit; // C2362
    }
    else
    { /*...*/ }

    int error_code = 42;

exit:
    return error_code;
}
```

É um bom estilo de programação usar as `break`, `continue` instruções, e `return` em vez da `goto` instrução, sempre que possível. No entanto, como a `break` instrução sai de apenas um nível de um loop, talvez seja necessário usar uma `goto` instrução para sair de um loop aninhado profundamente.

Para obter mais informações sobre rótulos e a `goto` instrução, consulte [instruções rotuladas](#).

Exemplo

Neste exemplo, uma `goto` instrução transfere o controle para o ponto rotulado `stop` quando `i` é igual a 3.

```
// goto_statement.cpp
#include <stdio.h>
int main()
{
    int i, j;

    for ( i = 0; i < 10; i++ )
    {
        printf_s( "Outer loop executing. i = %d\n", i );
        for ( j = 0; j < 2; j++ )
        {
            printf_s( " Inner loop executing. j = %d\n", j );
            if ( i == 3 )
                goto stop;
        }
    }

    // This message does not print:
    printf_s( "Loop exited. i = %d\n", i );

    stop:
    printf_s( "Jumped to stop. i = %d\n", i );
}
```

```
Outer loop executing. i = 0
Inner loop executing. j = 0
Inner loop executing. j = 1
Outer loop executing. i = 1
Inner loop executing. j = 0
Inner loop executing. j = 1
Outer loop executing. i = 2
Inner loop executing. j = 0
Inner loop executing. j = 1
Outer loop executing. i = 3
Inner loop executing. j = 0
Jumped to stop. i = 3
```

Confira também

[Instruções de atalho](#)

[Palavras-chave](#)

Transferências de controle

02/09/2020 • 2 minutes to read • [Edit Online](#)

Você pode usar a `goto` instrução ou um `case` rótulo em uma `switch` instrução para especificar um programa que ramificações após um inicializador. Esse código é inválido, a menos que a declaração que contém o inicializador esteja em um bloco delimitado pelo bloco em que a instrução de salto ocorre.

O exemplo a seguir mostra um loop que declara e inicializa os objetos `total`, `ch` e `i`. Há também uma instrução errada `goto` que transfere o controle após um inicializador.

```
// transfers_of_control.cpp
// compile with: /W1
// Read input until a nonnumeric character is entered.
int main()
{
    char MyArray[5] = {'2','2','a','c'};
    int i = 0;
    while( 1 )
    {
        int total = 0;

        char ch = MyArray[i++];

        if ( ch >= '0' && ch <= '9' )
        {
            goto Label1;

            int i = ch - '0';
        Label1:
            total += i;    // C4700: transfers past initialization of i.
        } // i would be destroyed here if goto error were not present
        else
            // Break statement transfers control out of loop,
            // destroying total and ch.
            break;
    }
}
```

No exemplo anterior, a `goto` instrução tenta transferir o controle após a inicialização de `i`. No entanto, se `i` tivesse sido declarado, mas não inicializado, a transferência seria válida.

Os objetos `total` e `ch`, declarados no bloco que serve como a *instrução* da `while` instrução, são destruídos quando esse bloco é encerrado usando a `break` instrução.

Namespaces (C++)

02/09/2020 • 13 minutes to read • [Edit Online](#)

Um namespace é uma região declarativa que fornece um escopo para os identificadores (os nomes de tipos, funções, variáveis, etc) dentro dele. Os namespaces são usados para organizar o código em grupos lógicos e para evitar colisões de nomes que podem ocorrer especialmente quando sua base de código inclui várias bibliotecas. Todos os identificadores no escopo do namespace são visíveis entre si sem qualificação. Os identificadores fora do namespace podem acessar os membros usando o nome totalmente qualificado para cada identificador, por exemplo `std::vector<std::string> vec;`, ou por uma [declaração using](#) para um único identificador (`using std::string`) ou uma [diretiva using](#) para todos os identificadores no namespace (`using namespace std;`). O código nos arquivos de cabeçalho sempre deve usar o nome totalmente qualificado do namespace.

O exemplo a seguir mostra uma declaração de namespace e três maneiras que o código fora do namespace pode acessar seus membros.

```
namespace ContosoData
{
    class ObjectManager
    {
    public:
        void DoSomething() {}
    };
    void Func(ObjectManager) {}
}
```

Use o nome totalmente qualificado:

```
ContosoData::ObjectManager mgr;
mgr.DoSomething();
ContosoData::Func(mgr);
```

Use uma declaração using para colocar um identificador no escopo:

```
using ContosoData::ObjectManager;
ObjectManager mgr;
mgr.DoSomething();
```

Use uma diretiva using para trazer tudo no namespace para o escopo:

```
using namespace ContosoData;

ObjectManager mgr;
mgr.DoSomething();
Func(mgr);
```

usando diretivas

A `using` diretiva permite que todos os nomes em um `namespace` sejam usados sem o *nome do namespace* como um qualificador explícito. Use uma diretiva `using` em um arquivo de implementação (ou seja, *.cpp) se você estiver usando vários identificadores diferentes em um namespace; Se você estiver usando apenas um ou dois identificadores, considere uma declaração `using` para colocar apenas esses identificadores no escopo e não

todos os identificadores no namespace. Se uma variável local tiver o mesmo nome de uma variável de namespace, a variável de namespace será oculta. É um erro ter uma variável de namespace com o mesmo nome de uma variável global.

NOTE

Uma diretiva `using` pode ser colocada na parte superior de um arquivo. `.cpp` (no escopo do arquivo) ou dentro de uma definição de classe ou função.

Em geral, evite colocar diretivas em arquivos de cabeçalho (`*.h`) porque qualquer arquivo que inclua esse cabeçalho colocará tudo no namespace no escopo, o que pode causar a ocultação de nomes e problemas de colisão de nomes que são muito difíceis de depurar. Sempre use nomes totalmente qualificados em um arquivo de cabeçalho. Se esses nomes forem muito longos, você poderá usar um alias de namespace para encurtá-los. (Veja abaixo.)

Declarando namespaces e membros de namespace

Normalmente, você declara um namespace em um arquivo de cabeçalho. Se as implementações de função estiverem em um arquivo separado, qualifique os nomes de função, como neste exemplo.

```
//contosoData.h
#pragma once
namespace ContosoDataServer
{
    void Foo();
    int Bar();
}
```

Implementações de função em `contosodata.cpp` devem usar o nome totalmente qualificado, mesmo que você coloque uma `using` diretiva na parte superior do arquivo:

```
#include "contosodata.h"
using namespace ContosoDataServer;

void ContosoDataServer::Foo() // use fully-qualified name here
{
    // no qualification needed for Bar()
    Bar();
}

int ContosoDataServer::Bar(){return 0;}
```

Um namespace pode ser declarado em vários blocos em um único arquivo e em vários arquivos. O compilador une as partes durante o pré-processamento e o namespace resultante contém todos os membros declarados em todas as partes. Um exemplo disso é o namespace padrão que é declarado em cada um dos arquivos de cabeçalho na biblioteca padrão.

Os membros de um namespace nomeado podem ser definidos fora do namespace em que são declarados pela qualificação explícita de nome que está sendo definido. No entanto, a definição deve aparecer após o ponto de declaração em um namespace que inclui o namespace da declaração. Por exemplo:

```
// defining_namespace_members.cpp
// C2039 expected
namespace V {
    void f();
}

void V::f() { }           // ok
void V::g() { }           // C2039, g() is not yet a member of V

namespace V {
    void g();
}
}
```

Esse erro pode ocorrer quando os membros do namespace são declarados em vários arquivos de cabeçalho, e você não incluiu esses cabeçalhos na ordem correta.

O namespace global

Se um identificador não for declarado em um namespace explícito, ele será parte do namespace global implícito. Em geral, tente evitar fazer declarações no escopo global quando possível, exceto para a [função principal](#) do ponto de entrada, que deve estar no namespace global. Para qualificar explicitamente um identificador global, use o operador de resolução de escopo sem nome, como em `::SomeFunction(x);`. Isso diferenciará o identificador de qualquer coisa com o mesmo nome em qualquer outro namespace e também ajudará a facilitar o seu código para outras pessoas entenderem.

O namespace std

Todas as funções e tipos de biblioteca padrão do C++ são declarados no `std` namespace ou namespaces aninhados dentro de `std`.

Namespaces aninhados

Os namespaces podem estar aninhados. Um namespace aninhado comum tem acesso não qualificado aos membros de seus pais, mas os membros pai não têm acesso não qualificado ao namespace aninhado (a menos que seja declarado como embutido), conforme mostrado no exemplo a seguir:

```
namespace ContosoDataServer
{
    void Foo();

    namespace Details
    {
        int CountImpl;
        void Bar() { return Foo(); }
    }

    int Baz(int i) { return Details::CountImpl; }
}
```

Namespaces aninhados comuns podem ser usados para encapsular detalhes de implementação internos que não fazem parte da interface pública do namespace pai.

Namespaces embutidos (C++ 11)

Ao contrário de um namespace aninhado comum, os membros de um namespace embutido são tratados como membros do namespace pai. Essa característica permite que a pesquisa dependente de argumento em funções

sobre carregadas funcionem em funções que têm sobre cargas em um namespace pai e aninhado embutido. Ele também permite que você declare uma especialização em um namespace pai para um modelo declarado no namespace embutido. O exemplo a seguir mostra como o código externo é associado ao namespace embutido por padrão:

```
//Header.h
#include <string>

namespace Test
{
    namespace old_ns
    {
        std::string Func() { return std::string("Hello from old"); }
    }

    inline namespace new_ns
    {
        std::string Func() { return std::string("Hello from new"); }
    }
}

#include "header.h"
#include <string>
#include <iostream>

int main()
{
    using namespace Test;
    using namespace std;

    string s = Func();
    std::cout << s << std::endl; // "Hello from new"
    return 0;
}
```

O exemplo a seguir mostra como você pode declarar uma especialização em um pai de um modelo que é declarado em um namespace embutido:

```
namespace Parent
{
    inline namespace new_ns
    {
        template <typename T>
        struct C
        {
            T member;
        };
    }
    template<>
    class C<int> {};
}
```

Você pode usar namespaces embutidos como um mecanismo de controle de versão para gerenciar alterações na interface pública de uma biblioteca. Por exemplo, você pode criar um único namespace pai e encapsular cada versão da interface em seu próprio namespace aninhado dentro do pai. O namespace que contém a versão mais recente ou preferida é qualificado como embutido e, portanto, é exposto como se fosse um membro direto do namespace pai. O código do cliente que invoca a classe pai:: será automaticamente associado ao novo código. Os clientes que preferem usar a versão mais antiga ainda podem acessá-lo usando o caminho totalmente qualificado para o namespace aninhado que tem esse código.

A palavra-chave `inline` deve ser aplicada à primeira declaração do namespace em uma unidade de compilação.

O exemplo a seguir mostra duas versões de uma interface, cada uma em um namespace aninhado. O `v_20` namespace tem alguma modificação da `v_10` interface e está marcado como embutido. O código do cliente que usa a nova biblioteca e as chamadas `Contoso::Funcs::Add` invocará a versão `v_20`. O código que tenta chamar `Contoso::Funcs::Divide` agora receberá um erro de tempo de compilação. Se eles realmente precisarem dessa função, eles ainda poderão acessar a `v_10` versão chamando explicitamente `Contoso::v_10::Funcs::Divide`.

```
namespace Contoso
{
    namespace v_10
    {
        template <typename T>
        class Funcs
        {
        public:
            Funcs(void);
            T Add(T a, T b);
            T Subtract(T a, T b);
            T Multiply(T a, T b);
            T Divide(T a, T b);
        };
    }

    inline namespace v_20
    {
        template <typename T>
        class Funcs
        {
        public:
            Funcs(void);
            T Add(T a, T b);
            T Subtract(T a, T b);
            T Multiply(T a, T b);
            std::vector<double> Log(double);
            T Accumulate(std::vector<T> nums);
        };
    }
}
```

Aliases de namespace

Os nomes de namespace precisam ser exclusivos, o que significa que, geralmente, eles não devem ser curtos demais. Se o comprimento de um nome dificulta a leitura do código ou é entediante digitar em um arquivo de cabeçalho no qual as diretivas não podem ser usadas, você pode criar um alias de namespace que serve como uma abreviação do nome real. Por exemplo:

```
namespace a_very_long_namespace_name { class Foo {}; }
namespace AVLNN = a_very_long_namespace_name;
void Bar(AVLNN::Foo foo){ }
```

namespaces anônimos ou não nomeados

Você pode criar um namespace explícito, mas não dar um nome a ele:

```
namespace
{
    int MyFunc(){}
}
```

Isso é chamado de namespace não nomeado ou anônimo e é útil quando você deseja tornar declarações de variáveis invisíveis para código em outros arquivos (ou seja, dar a elas vínculos internos) sem precisar criar um namespace nomeado. Todo o código no mesmo arquivo pode ver os identificadores em um namespace sem nome, mas os identificadores, juntamente com o próprio namespace, não são visíveis fora desse arquivo — ou mais precisamente fora da unidade de tradução.

Confira também

[Declarações e definições](#)

Enumerações (C++)

02/09/2020 • 9 minutes to read • [Edit Online](#)

Uma enumeração é um tipo definido pelo usuário que consiste em um conjunto de constantes integras nomeadas que são conhecidas como enumeradores.

NOTE

Este artigo aborda o tipo de linguagem ISO Standard C++ `enum` e o tipo de **classe de enumeração** com escopo definido (ou fortemente tipado), que é introduzido no C++ 11. Para obter informações sobre a **classe de enumeração pública** ou tipos de **classe de enumeração privada** em c++/CLI e c++/CX, consulte [enum Class](#).

Sintaxe

```
// unscoped enum:  
enum [identifier] [: type]  
{enum-list};  
  
// scoped enum:  
enum [class|struct]  
[identifier] [: type]  
{enum-list};
```

```
// Forward declaration of enumerations (C++11):  
enum A : int; // non-scoped enum must have type specified  
enum class B; // scoped enum defaults to int but ...  
enum class C : short; // ... may have any integral underlying type
```

parâmetros

ID

O nome do tipo dado à enumeração.

tipo

O tipo subjacente dos enumeradores; todos os enumeradores têm o mesmo tipo subjacente. Pode ser qualquer tipo integral.

lista de enums

Uma lista separada por vírgulas dos enumeradores na enumeração. Cada enumerador ou nome de variável no escopo deve ser exclusivo. No entanto, os valores podem ser duplicados. Em uma enumeração sem escopo, o escopo é o escopo ao redor; em uma enumeração com escopo, o escopo é a própria *lista de enums*. Em uma enumeração com escopo, a lista pode estar vazia, o que, em vigor, define um novo tipo integral.

class

Ao usar essa palavra-chave na declaração, você especifica o escopo da enumeração e um *identificador* deve ser fornecido. Você também pode usar a `struct` palavra-chave no lugar de `class`, pois elas são semanticamente equivalentes nesse contexto.

Escopo do enumerador

Uma enumeração fornece o contexto para descrever um intervalo de valores que são representados como constantes nomeadas e que também são chamados de enumeradores. Nos tipos de enum originais de C e C++, os enumeradores não qualificados são visíveis em todo o escopo no qual o enum é declarado. Em enums com escopo, o nome do enumerador deve ser qualificado pelo nome do tipo enum. O exemplo a seguir demonstra essa diferença básica entre os dois tipos de enums:

```
namespace CardGame_Scoped
{
    enum class Suit { Diamonds, Hearts, Clubs, Spades };

    void PlayCard(Suit suit)
    {
        if (suit == Suit::Clubs) // Enumerator must be qualified by enum type
        { /*...*/}
    }
}

namespace CardGame_NonScoped
{
    enum Suit { Diamonds, Hearts, Clubs, Spades };

    void PlayCard(Suit suit)
    {
        if (suit == Clubs) // Enumerator is visible without qualification
        { /*...*/}
    }
}
```

Cada nome em uma enumeração recebe um valor integral que corresponde ao seu local na ordem dos valores na enumeração. Por padrão, o primeiro valor é atribuído a 0, o seguinte é atribuído a 1, e assim por diante, mas você pode definir explicitamente o valor de um enumerador, como mostrado aqui:

```
enum Suit { Diamonds = 1, Hearts, Clubs, Spades };
```

O enumerador `Diamonds` recebe o valor `1`. Os enumeradores subsequentes, se não recebem um valor explícito, recebem o valor do enumerador anterior mais um. No exemplo anterior, `Hearts` teria o valor 2, `Clubs` teria 3 e assim por diante.

Cada enumerador é tratado como uma constante e deve ter um nome exclusivo dentro do escopo em que o `enum` é definido (para enums sem escopo) ou dentro `enum` dele mesmo (para enumerações com escopo). Os valores dados para os nomes não devem ser exclusivos. Por exemplo, se a declaração de um enum sem escopo `Suit` for esta:

```
enum Suit { Diamonds = 5, Hearts, Clubs = 4, Spades };
```

Os valores de `Diamonds`, `Hearts`, `Clubs` e `Spades` são 5, 6, 4, e 5, respectivamente. Observe que 5 será usado mais de uma vez; isso é permitido mesmo que não seja pretendido. Essas regras são as mesmas para enums com escopo.

Regras de conversão

As constantes de enumeração sem escopo podem ser convertidas implicitamente em `int`, mas uma `int` nunca é conversível implicitamente em um valor de enumeração. O exemplo a seguir mostra o que acontece se você tentar atribuir a `hand` um valor que não seja `Suit`:

```
int account_num = 135692;
Suit hand;
hand = account_num; // error C2440: '=' : cannot convert from 'int' to 'Suit'
```

Uma conversão é necessária para converter um `int` para um enumerador com escopo ou sem escopo. No entanto, você pode promover um enumerador para um valor inteiro sem uma conversão.

```
int account_num = Hearts; //OK if Hearts is in a unscoped enum
```

O uso de conversões implícitas dessa maneira pode levar a efeitos colaterais não intencionais. Para ajudar a eliminar erros de programação associados aos enums sem escopo, os valores enum com escopo são fortemente tipados. Os enumeradores com escopo devem ser qualificados pelo nome do tipo enum (identificador) e não podem ser convertidos implicitamente, conforme mostrado no exemplo a seguir:

```
namespace ScopedEnumConversions
{
    enum class Suit { Diamonds, Hearts, Clubs, Spades };

    void AttemptConversions()
    {
        Suit hand;
        hand = Clubs; // error C2065: 'Clubs' : undeclared identifier
        hand = Suit::Clubs; //Correct.
        int account_num = 135692;
        hand = account_num; // error C2440: '=' : cannot convert from 'int' to 'Suit'
        hand = static_cast<Suit>(account_num); // OK, but probably a bug!!!

        account_num = Suit::Hearts; // error C2440: '=' : cannot convert from 'Suit' to 'int'
        account_num = static_cast<int>(Suit::Hearts); // OK
    }
}
```

Observe que a linha `hand = account_num;` ainda causa o erro que ocorre com enums sem escopo, como mostrado anteriormente. É permitido com uma conversão explícita. No entanto, com enums com escopo, a conversão tentada na próxima instrução, `account_num = Suit::Hearts;`, não é mais permitida sem uma conversão explícita.

Enums sem enumeradores

Visual Studio 2017 versão 15,3 e posterior (disponível com [/std:c++ 17](#)): ao definir uma enumeração (regular ou com escopo) com um tipo subjacente explícito e nenhum enumerador, você pode, na verdade, introduzir um novo tipo integral que não tenha nenhuma conversão implícita para nenhum outro tipo. Ao usar esse tipo em vez de seu tipo subjacente interno, você pode eliminar o potencial para erros sutis causados por conversões implícitas inadvertidas.

```
enum class byte : unsigned char { };
```

O novo tipo é uma cópia exata do tipo subjacente e, portanto, tem a mesma convenção de chamada, o que significa que ele pode ser usado em ABIs sem nenhuma penalidade de desempenho. Nenhuma conversão é necessária quando as variáveis do tipo são inicializadas usando a inicialização da lista direta. O exemplo a seguir mostra como inicializar enums sem enumeradores em vários contextos:

```
enum class byte : unsigned char { };

enum class E : int { };
E e1{ 0 };
E e2 = E{ 0 };

struct X
{
    E e{ 0 };
    X() : e{ 0 } { }
};

E* p = new E{ 0 };

void f(E e) {}

int main()
{
    f(E{ 0 });
    byte i{ 42 };
    byte j = byte{ 42 };

    // unsigned char c = j; // C2440: 'initializing': cannot convert from 'byte' to 'unsigned char'
    return 0;
}
```

Confira também

[Declarações de enumeração C](#)

[Palavras-chave](#)

NOTE

No C++ 17 e posterior, a `std::variant` class é uma alternativa de tipo seguro para um union .

Um `union` é um tipo definido pelo usuário no qual todos os membros compartilham o mesmo local de memória. Essa definição significa que, a qualquer momento, um union não pode conter mais de um objeto de sua lista de membros. Isso também significa que, independentemente de quantos membros a union têm, ele sempre usa apenas memória suficiente para armazenar o maior membro.

Um union pode ser útil para conservar memória quando você tem muitos objetos e memória limitada. No entanto, um union requer um cuidado extra com o uso correto. Você é responsável por garantir que sempre acesse o mesmo membro que você atribuiu. Se qualquer tipo de membro tiver uma con não trivial struct ou, então, você deverá escrever código adicional para con explicitamente struct e destruir esse membro. Antes de usar um union , considere se o problema que você está tentando resolver pode ser melhor expresso com o uso de class tipos base e derivados class .

Sintaxe

```
union *** tag * opt { aceitar * member-list *** };
```

Parâmetros

`tag`

O nome do tipo fornecido para o union .

`member-list`

Membros que o union pode conter.

Declarar um union

Inicie a declaração de um union usando a `union` palavra-chave e coloque a lista de membros entre chaves:

```
// declaring_a_union.cpp
union RecordType // Declare a simple union type
{
    char    ch;
    int     i;
    long    l;
    float   f;
    double  d;
    int *int_ptr;
};

int main()
{
    RecordType t;
    t.i = 5; // t holds an int
    t.f = 7.25; // t now holds a float
}
```

Usar um union

No exemplo anterior, qualquer código que acessa o union precisa saber qual membro contém os dados. A solução mais comum para esse problema é chamada de *discriminado union *. Ele engloba o union em struct e inclui um enum membro que indica o tipo de membro atualmente armazenado no union . O exemplo a seguir mostra o padrão básico:

```
#include <queue>

using namespace std;

enum class WeatherDataType
{
    Temperature, Wind
};

struct TempData
{
    int StationId;
    time_t time;
    double current;
    double max;
    double min;
};

struct WindData
{
    int StationId;
    time_t time;
    int speed;
    short direction;
};

struct Input
{
    WeatherDataType type;
    union
    {
        TempData temp;
        WindData wind;
    };
};

// Functions that are specific to data types
void Process_Temp(TempData t) {}
void Process_Wind(WindData w) {}

void Initialize(std::queue<Input>& inputs)
{
    Input first;
    first.type = WeatherDataType::Temperature;
    first.temp = { 101, 1418855664, 91.8, 108.5, 67.2 };
    inputs.push(first);

    Input second;
    second.type = WeatherDataType::Wind;
    second.wind = { 204, 1418859354, 14, 27 };
    inputs.push(second);
}

int main(int argc, char* argv[])
{
    // Container for all the data records
    queue<Input> inputs;
    Initialize(inputs);
    while (!inputs.empty())
    {
```

```

    {
        Input const i = inputs.front();
        switch (i.type)
        {
            case WeatherDataType::Temperature:
                Process_Temp(i.temp);
                break;
            case WeatherDataType::Wind:
                Process_Wind(i.wind);
                break;
            default:
                break;
        }
        inputs.pop();
    }
    return 0;
}

```

No exemplo anterior, o union no `Input` struct não tem nome, portanto, ele é chamado de *anônimo union*. Seus membros podem ser acessados diretamente como se fossem membros do struct. Para obter mais informações sobre como usar um anônimo union, consulte a [seção union anônima](#).

O exemplo anterior mostra um problema que você também pode resolver usando class tipos que derivam de uma base comum class. Você pode ramificar seu código com base no tipo de tempo de execução de cada objeto no contêiner. Seu código pode ser mais fácil de manter e entender, mas também pode ser mais lento do que usar um union. Além disso, com um union, você pode armazenar tipos não relacionados. Um union permite que você altere dinamicamente o tipo do valor armazenado sem alterar o tipo da union variável em si. Por exemplo, você pode criar uma matriz heterogênea de `MyUnionType`, cujos elementos armazenam valores diferentes de tipos diferentes.

É fácil refazer o uso indevido do `Input` struct exemplo. Cabe ao usuário usar o discriminador corretamente para acessar o membro que contém os dados. Você pode se proteger contra uso indevido, fazendo o union `private` e fornecendo funções de acesso especiais, conforme mostrado no exemplo a seguir.

Irrestrito union (c++ 11)

No C++ 03 e versões anteriores, um union pode conter membros que não são de static dados que têm um class tipo, desde que o tipo não tenha um usuário que tenha fornecido o struct ORS, de struct ORS nem de operadores de atribuição. No C++ 11, essas restrições são removidas. Se você incluir tal membro em seu union, o compilador marcará automaticamente qualquer função de membro especial que não seja fornecida pelo usuário como `deleted`. Se o union for um anônimo union dentro de um class ou struct, qualquer função de membro especial do class ou struct que não seja fornecida pelo usuário será marcada como `deleted`. O exemplo a seguir mostra como lidar com esse caso. Um dos membros do union tem um membro que exige esse tratamento especial:

```

// for MyVariant
#include <crtdbg.h>
#include <new>
#include <utility>

// for sample objects and output
#include <string>
#include <vector>
#include <iostream>

using namespace std;

struct A
{
    A() = default;

```

```
    ...
    A(int i, const string& str) : num(i), name(str) {}

    int num;
    string name;
    //...
};

struct B
{
    B() = default;
    B(int i, const string& str) : num(i), name(str) {}

    int num;
    string name;
    vector<int> vec;
    // ...
};

enum class Kind { None, A, B, Integer };

#pragma warning (push)
#pragma warning(disable:4624)
class MyVariant
{
public:
    MyVariant()
        : kind_(Kind::None)
    {
    }

    MyVariant(Kind kind)
        : kind_(kind)
    {
        switch (kind_)
        {
        case Kind::None:
            break;
        case Kind::A:
            new (&a_) A();
            break;
        case Kind::B:
            new (&b_) B();
            break;
        case Kind::Integer:
            i_ = 0;
            break;
        default:
            _ASSERT(false);
            break;
        }
    }

    ~MyVariant()
    {
        switch (kind_)
        {
        case Kind::None:
            break;
        case Kind::A:
            a_.~A();
            break;
        case Kind::B:
            b_.~B();
            break;
        case Kind::Integer:
            break;
        default:
            _ASSERT(false);
            break;
        }
    }
};
```

```

        break;
    }
    kind_ = Kind::None;
}

MyVariant(const MyVariant& other)
: kind_(other.kind_)
{
    switch (kind_)
    {
        case Kind::None:
            break;
        case Kind::A:
            new (&a_) A(other.a_);
            break;
        case Kind::B:
            new (&b_) B(other.b_);
            break;
        case Kind::Integer:
            i_ = other.i_;
            break;
        default:
            _ASSERT(false);
            break;
    }
}

MyVariant(MyVariant&& other)
: kind_(other.kind_)
{
    switch (kind_)
    {
        case Kind::None:
            break;
        case Kind::A:
            new (&a_) A(move(other.a_));
            break;
        case Kind::B:
            new (&b_) B(move(other.b_));
            break;
        case Kind::Integer:
            i_ = other.i_;
            break;
        default:
            _ASSERT(false);
            break;
    }
    other.kind_ = Kind::None;
}

MyVariant& operator=(const MyVariant& other)
{
    if (&other != this)
    {
        switch (other.kind_)
        {
            case Kind::None:
                this->~MyVariant();
                break;
            case Kind::A:
                *this = other.a_;
                break;
            case Kind::B:
                *this = other.b_;
                break;
            case Kind::Integer:
                *this = other.i_;
                break;
            default:
                break;
        }
    }
}

```

```

        _ASSERT(false);
        break;
    }
}

return *this;
}

MyVariant& operator=(MyVariant&& other)
{
    _ASSERT(this != &other);
    switch (other.kind_)
    {
        case Kind::None:
            this->~MyVariant();
            break;
        case Kind::A:
            *this = move(other.a_);
            break;
        case Kind::B:
            *this = move(other.b_);
            break;
        case Kind::Integer:
            *this = other.i_;
            break;
        default:
            _ASSERT(false);
            break;
    }
    other.kind_ = Kind::None;
    return *this;
}

MyVariant(const A& a)
: kind_(Kind::A), a_(a)
{
}

MyVariant(A&& a)
: kind_(Kind::A), a_(move(a))
{
}

MyVariant& operator=(const A& a)
{
    if (kind_ != Kind::A)
    {
        this->~MyVariant();
        new (this) MyVariant(a);
    }
    else
    {
        a_ = a;
    }
    return *this;
}

MyVariant& operator=(A&& a)
{
    if (kind_ != Kind::A)
    {
        this->~MyVariant();
        new (this) MyVariant(move(a));
    }
    else
    {
        a_ = move(a);
    }
    return *this;
}

```

```

MyVariant(const B& b)
    : kind_(Kind::B), b_(b)
{
}

MyVariant(B&& b)
    : kind_(Kind::B), b_(move(b))
{
}

MyVariant& operator=(const B& b)
{
    if (kind_ != Kind::B)
    {
        this->~MyVariant();
        new (this) MyVariant(b);
    }
    else
    {
        b_ = b;
    }
    return *this;
}

MyVariant& operator=(B&& b)
{
    if (kind_ != Kind::B)
    {
        this->~MyVariant();
        new (this) MyVariant(move(b));
    }
    else
    {
        b_ = move(b);
    }
    return *this;
}

MyVariant(int i)
    : kind_(Kind::Integer), i_(i)
{
}

MyVariant& operator=(int i)
{
    if (kind_ != Kind::Integer)
    {
        this->~MyVariant();
        new (this) MyVariant(i);
    }
    else
    {
        i_ = i;
    }
    return *this;
}

Kind GetKind() const
{
    return kind_;
}

A& GetA()
{
    _ASSERT(kind_ == Kind::A);
    return a_;
}

```

```

const A& GetA() const
{
    _ASSERT(kind_ == Kind::A);
    return a_;
}

B& GetB()
{
    _ASSERT(kind_ == Kind::B);
    return b_;
}

const B& GetB() const
{
    _ASSERT(kind_ == Kind::B);
    return b_;
}

int& GetInteger()
{
    _ASSERT(kind_ == Kind::Integer);
    return i_;
}

const int& GetInteger() const
{
    _ASSERT(kind_ == Kind::Integer);
    return i_;
}

private:
    Kind kind_;
    union
    {
        A a_;
        B b_;
        int i_;
    };
};

#pragma warning (pop)

int main()
{
    A a(1, "Hello from A");
    B b(2, "Hello from B");

    MyVariant mv_1 = a;

    cout << "mv_1 = a: " << mv_1.GetA().name << endl;
    mv_1 = b;
    cout << "mv_1 = b: " << mv_1.GetB().name << endl;
    mv_1 = A(3, "hello again from A");
    cout << R"aaa(mv_1 = A(3, "hello again from A"): )aaa" << mv_1.GetA().name << endl;
    mv_1 = 42;
    cout << "mv_1 = 42: " << mv_1.GetInteger() << endl;

    b.vec = { 10,20,30,40,50 };

    mv_1 = move(b);
    cout << "After move, mv_1 = b: vec.size = " << mv_1.GetB().vec.size() << endl;

    cout << endl << "Press a letter" << endl;
    char c;
    cin >> c;
}

```

Um union não pode armazenar uma referência. Um union também não dá suporte à herança. Isso significa que

você não pode usar um union como base class , ou herdar de outro class , ou ter funções virtuais.

Inicializar um union

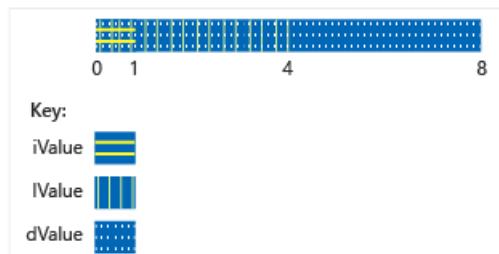
Você pode declarar e inicializar um union na mesma instrução atribuindo uma expressão entre chaves. A expressão é avaliada e atribuída ao primeiro campo do union .

```
#include <iostream>
using namespace std;

union NumericType
{
    short      iValue;
    long       lValue;
    double     dValue;
};

int main()
{
    union NumericType Values = { 10 };    // iValue = 10
    cout << Values.iValue << endl;
    Values.dValue = 3.1416;
    cout << Values.dValue << endl;
}
/* Output:
10
3.141600
*/
```

O `NumericType` union é organizado na memória (conceitualmente), conforme mostrado na figura a seguir.



Armazenamento de dados em um `NumericType` union

Modo union

Um anônimo union é um declarado sem um `class-name` ou `declarator-List` .

```
union { member-list }
```

Nomes declarados em um anônimo union são usados diretamente, como variáveis não membro. Isso implica que os nomes declarados em um anônimo union devem ser exclusivos no escopo ao redor.

Um anônimo union está sujeito a essas restrições adicionais:

- Se declarado em escopo de arquivo ou de namespace, ele também deve ser declarado como `static` .
- Ele só pode ter `public` Membros; ter `private` e `protected` Membros em um anônimo union geram erros.
- Ele não pode ter funções de membro.

Consulte também

[Classes e structs](#)

[Palavras-chave](#)

[class](#)

[struct](#)

Funções (C++)

02/09/2020 • 24 minutes to read • [Edit Online](#)

Uma função é um bloco de código que executa alguma operação. Uma função pode, opcionalmente, definir parâmetros de entrada que permitem aos chamadores passar argumentos para a função. Uma função pode, opcionalmente, retornar um valor como saída. As funções são úteis para encapsular operações comuns em um único bloco reutilizável, idealmente com um nome que descreve claramente o que a função faz. A função a seguir aceita dois inteiros de um chamador e retorna sua soma; *a* e *b* são *parâmetros* do tipo `int`.

```
int sum(int a, int b)
{
    return a + b;
}
```

A função pode ser invocada, ou *chamada*, de qualquer número de locais no programa. Os valores que são passados para a função são os *argumentos*, cujos tipos devem ser compatíveis com os tipos de parâmetro na definição de função.

```
int main()
{
    int i = sum(10, 32);
    int j = sum(i, 66);
    cout << "The value of j is" << j << endl; // 108
}
```

Não há nenhum limite prático para o comprimento da função, mas bom design visa funções que executam uma única tarefa bem definida. Algoritmos complexos devem ser divididos em funções simples e fáceis de entender, sempre que possível.

As funções definidas no escopo de classe são chamadas de funções de membro. Em C++, ao contrário de outras linguagens, uma função também pode ser definida no escopo do namespace (incluindo o namespace global implícito). Essas funções são chamadas de funções *gratuitas* ou *de funções que não são de membro*. Eles são usados extensivamente na biblioteca padrão.

As funções podem estar *sobre carregadas*, o que significa que versões diferentes de uma função podem compartilhar o mesmo nome se diferirem pelo número e/ou tipo de parâmetros formais. Para obter mais informações, consulte [sobrecarga de função](#).

Partes de uma declaração de função

Uma *declaração* de função mínima consiste no tipo de retorno, nome da função e lista de parâmetros (que pode estar vazio), juntamente com palavras-chave opcionais que fornecem instruções adicionais ao compilador. O exemplo a seguir é uma declaração de função:

```
int sum(int a, int b);
```

Uma definição de função consiste em uma declaração, mais o *corpo*, que é todo o código entre as chaves:

```
int sum(int a, int b)
{
    return a + b;
}
```

Uma declaração de função seguida por um ponto-e-vírgula pode aparecer em vários lugares em um programa. Ele deve aparecer antes de qualquer chamada para essa função em cada unidade de tradução. A definição da função deve aparecer apenas uma vez no programa, de acordo com a regra de definição única (ODR).

As partes necessárias de uma declaração de função são:

1. O tipo de retorno, que especifica o tipo do valor que a função retorna, ou `void` se nenhum valor for retornado. No C++ 11, `auto` é um tipo de retorno válido que instrui o compilador a inferir o tipo da instrução `return`. No C++ 14, `decltype(auto)` também é permitido. Para obter mais informações, consulte dedução de tipo em tipos de retorno abaixo.
2. O nome da função, que deve começar com uma letra ou sublinhado e não pode conter espaços. Em geral, os sublinhados à esquerda nos nomes de função de biblioteca padrão indicam funções de membro privado ou funções não-membro que não se destinam ao uso pelo seu código.
3. A lista de parâmetros, uma chave delimitada, um conjunto separado por vírgulas de zero ou mais parâmetros que especificam o tipo e, opcionalmente, um nome local pelo qual os valores podem ser acessados dentro do corpo da função.

As partes opcionais de uma declaração de função são:

1. `constexpr`, que indica que o valor de retorno da função é um valor constante que pode ser calculado em tempo de compilação.

```
constexpr float exp(float x, int n)
{
    return n == 0 ? 1 :
        n % 2 == 0 ? exp(x * x, n / 2) :
        exp(x * x, (n - 1) / 2) * x;
};
```

2. Sua especificação de vinculação `extern` ou `static`.

```
//Declare printf with C linkage.
extern "C" int printf( const char *fmt, ... );
```

Para obter mais informações, consulte [unidades de tradução e ligação](#).

3. `inline`, que instrui o compilador a substituir cada chamada à função pelo próprio código de função. O inalinhamento pode ajudar o desempenho em cenários em que uma função é executada rapidamente e é invocada repetidamente em uma seção de código crítica de desempenho.

```
inline double Account::GetBalance()
{
    return balance;
}
```

Para obter mais informações, consulte [funções embutidas](#).

4. Uma `noexcept` expressão, que especifica se a função pode ou não gerar uma exceção. No exemplo a seguir,

a função não lançará uma exceção se a `is_pod` expressão for avaliada como `true`.

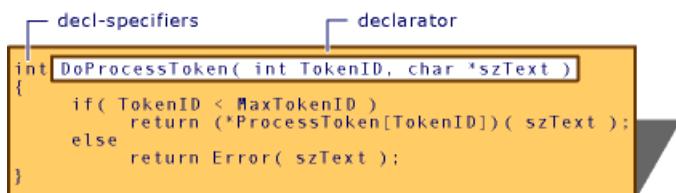
```
#include <type_traits>

template <typename T>
T copy_object(T& obj) noexcept(std::is_pod<T>) {...}
```

Para obter mais informações, consulte [noexcept](#).

5. (Somente funções de membro) Os qualificadores CV, que especificam se a função é `const` ou `volatile`.
6. (Somente funções de membro) `virtual`, `override` ou `final`. `virtual` Especifica que uma função pode ser substituída em uma classe derivada. `override` significa que uma função em uma classe derivada está substituindo uma função virtual. `final` significa que uma função não pode ser substituída em nenhuma classe derivada adicional. Para obter mais informações, consulte [funções virtuais](#).
7. (somente funções de membro) `static` aplicado a uma função de membro significa que a função não está associada a nenhuma instância de objeto da classe.
8. (Somente funções membro não estáticas) O qualificador de referência, que especifica para o compilador qual sobrecarga de uma função escolher quando o parâmetro de objeto implícito (`*this`) é uma referência rvalue versus uma referência lvalue. Para obter mais informações, consulte [sobrecarga de função](#).

A figura a seguir mostra as partes de uma definição de função. A área sombreada é o corpo da função.



Partes de uma definição de função

Definições de função

Uma *definição de função* consiste na declaração e no corpo da função, entre chaves, que contém declarações de variáveis, instruções e expressões. O exemplo a seguir mostra uma definição de função completa:

```
int foo(int i, std::string s)  
{  
    int value {i};  
    MyClass mc;  
    if(strcmp(s, "default") != 0)  
    {  
        value = mc.do_something(i);  
    }  
    return value;  
}
```

Variáveis declaradas dentro do corpo são chamadas de variáveis locais ou locais. Eles saem do escopo quando a função é encerrada; Portanto, uma função nunca deve retornar uma referência a um local!

```
 MyClass& boom(int i, std::string s)
{
    int value {i};
    MyClass mc;
    mc.Initialize(i,s);
    return mc;
}
```

funções const e constexpr

Você pode declarar uma função de membro como `const` para especificar que a função não tem permissão para alterar os valores de quaisquer membros de dados na classe. Ao declarar uma função de membro como `const`, você ajuda o compilador a impor a *exatidão da constante*. Se alguém tentar modificar o objeto por engano usando uma função declarada como `const`, um erro do compilador será gerado. Para obter mais informações, consulte [const](#).

Declare uma função como `constexpr` quando o valor que ele produz pode ser determinado em tempo de compilação. Uma função `constexpr` geralmente é executada mais rapidamente do que uma função regular. Para obter mais informações, consulte [constexpr](#).

Modelos de função

Um modelo de função é semelhante a um modelo de classe; Ele gera funções concretas com base nos argumentos do modelo. Em muitos casos, o modelo é capaz de inferir os argumentos de tipo e, portanto, não é necessário especificá-los explicitamente.

```
template<typename Lhs, typename Rhs>
auto Add2(const Lhs& lhs, const Rhs& rhs)
{
    return lhs + rhs;
}

auto a = Add2(3.13, 2.895); // a is a double
auto b = Add2(string{ "Hello" }, string{ " World" }); // b is a std::string
```

Para obter mais informações, consulte [modelos de função](#)

Parâmetros de função e argumentos

Uma função tem uma lista de parâmetros separados por vírgulas de zero ou mais tipos, cada um com um nome pelo qual ele pode ser acessado dentro do corpo da função. Um modelo de função pode especificar parâmetros de tipo ou valor adicionais. O chamador passa argumentos, que são valores concretos cujos tipos são compatíveis com a lista de parâmetros.

Por padrão, os argumentos são passados para a função por valor, o que significa que a função recebe uma cópia do objeto que está sendo passado. Para objetos grandes, fazer uma cópia pode ser caro e nem sempre é necessário. Para fazer com que os argumentos sejam passados por referência (especificamente referência `lvalue`), adicione um quantificador de referência ao parâmetro:

```
void DoSomething(std::string& input){...}
```

Quando uma função modifica um argumento que é passado por referência, ela modifica o objeto original, não uma cópia local. Para impedir que uma função modifique tal argumento, qualifique o parâmetro como `const&`:

```
void DoSomething(const std::string& input){...}
```

C++ 11: Para manipular explicitamente argumentos que são passados por rvalue-Reference ou lvalue-Reference, use um e comercial duplo no parâmetro para indicar uma referência universal:

```
void DoSomething(const std::string&& input){...}
```

Uma função declarada com a palavra-chave **Single** `void` na lista de declarações de parâmetro não usa argumentos, desde que a palavra-chave `void` seja o primeiro e único membro da lista de declarações de argumento. Os argumentos do tipo `void` em outro lugar na lista produzem erros. Por exemplo:

```
// OK same as GetTickCount()
long GetTickCount( void );
```

Observe que, embora seja ilegal especificar um argumento, `void` exceto conforme descrito aqui, os tipos derivados do tipo `void` (como ponteiros `void` e matrizes de `void`) podem aparecer em qualquer lugar da lista de declaração de argumento.

Argumentos padrão

O último parâmetro ou parâmetros em uma assinatura de função pode ser atribuído a um argumento padrão, o que significa que o chamador pode deixar o argumento ao chamar a função, a menos que queira especificar algum outro valor.

```
int DoSomething(int num,
    string str,
    Allocator& alloc = defaultAllocator)
{ ... }

// OK both parameters are at end
int DoSomethingElse(int num,
    string str = string{ "Working" },
    Allocator& alloc = defaultAllocator)
{ ... }

// C2548: 'DoMore': missing default parameter for parameter 2
int DoMore(int num = 5, // Not a trailing parameter!
    string str,
    Allocator& = defaultAllocator)
{...}
```

Para obter mais informações, consulte [argumentos padrão](#).

Tipos de retorno de função

Uma função pode não retornar outra função ou uma matriz interna; no entanto, ele pode retornar ponteiros para esses tipos ou um *lambda*, que produz um objeto de função. Exceto para esses casos, uma função pode retornar um valor de qualquer tipo que esteja no escopo ou pode não retornar nenhum valor; nesse caso, o tipo de retorno é `void`.

Tipos de retorno à direita

Um tipo de retorno "comum" está localizado no lado esquerdo da assinatura da função. Um *tipo de retorno à direita* está localizado no lado direito da assinatura e é precedido pelo `->` operador. Os tipos de retorno à direita são especialmente úteis em modelos de função quando o tipo de valor de retorno depende de parâmetros de

modelo.

```
template<typename Lhs, typename Rhs>
auto Add(const Lhs& lhs, const Rhs& rhs) -> decltype(lhs + rhs)
{
    return lhs + rhs;
}
```

Quando `auto` é usado em conjunto com um tipo de retorno à direita, ele serve apenas como um espaço reservado para qualquer coisa que a expressão `decltype` produz e não executa a dedução de tipo.

Variáveis locais de função

Uma variável que é declarada dentro de um corpo de função é chamada de *variável local* ou simplesmente um *local*. Locais não estáticos só são visíveis dentro do corpo da função e, se eles forem declarados na pilha, saem do escopo quando a função é encerrada. Quando você constrói uma variável local e a retorna por valor, o compilador geralmente pode executar a *otimização do valor de retorno nomeado* para evitar operações de cópia desnecessárias. Se você retornar uma variável local por referência, o compilador emitirá um aviso porque qualquer tentativa feita pelo chamador para usar essa referência ocorrerá depois que o local for destruído.

Em C++, uma variável local pode ser declarada como estática. A variável só é visível dentro do corpo da função, mas existe uma única cópia da variável para todas as instâncias da função. Os objetos estáticos locais são destruídos durante o término especificado por `atexit`. Se um objeto estático não foi construído porque o fluxo de programa do controle ignorou a declaração dele, nenhuma tentativa de destruição de objeto será feita.

Dedução de tipo em tipos de retorno (C++ 14)

No C++ 14, você pode usar `auto` o para instruir o compilador a inferir o tipo de retorno do corpo da função sem precisar fornecer um tipo de retorno à direita. Observe que `auto` sempre deduz a um retorno por valor. Use `auto&` para instruir o compilador a deduzir uma referência.

Neste exemplo, `auto` será deduzido como uma cópia de valor não const da soma de LHS e RHS.

```
template<typename Lhs, typename Rhs>
auto Add2(const Lhs& lhs, const Rhs& rhs)
{
    return lhs + rhs; //returns a non-const object by value
}
```

Observe que `auto` o não preserva a constante qualidade do tipo que ele deduz. Para funções de encaminhamento cujo valor de retorno precisa preservar o const-qualidade ou ref-qualidade de seus argumentos, você pode usar a `decltype(auto)` palavra-chave, que usa as `decltype` regras de inferência de tipos e preserva todas as informações de tipo. `decltype(auto)` pode ser usado como um valor de retorno comum no lado esquerdo ou como um valor de retorno à direita.

O exemplo a seguir (com base no código de [N3493](#)), mostra `decltype(auto)` ser usado para habilitar o encaminhamento perfeito de argumentos de função em um tipo de retorno que não é conhecido até que o modelo seja instanciado.

```

template<typename F, typename Tuple = tuple<T...>, int... I>
decltype(auto) apply_(F&& f, Tuple&& args, index_sequence<I...>)
{
    return std::forward<F>(f)(std::get<I>(std::forward<Tuple>(args))...);
}

template<typename F, typename Tuple = tuple<T...>,
         typename Indices = make_index_sequence<tuple_size<Tuple>::value >>
decltype( auto)
apply(F&& f, Tuple&& args)
{
    return apply_(std::forward<F>(f), std::forward<Tuple>(args), Indices());
}

```

Retornando vários valores de uma função

Há várias maneiras de retornar mais de um valor de uma função:

1. Encapsula os valores em uma classe nomeada ou objeto de struct. Requer que a definição de classe ou estrutura seja visível para o chamador:

```

#include <string>
#include <iostream>

using namespace std;

struct S
{
    string name;
    int num;
};

S g()
{
    string t{ "hello" };
    int u{ 42 };
    return { t, u };
}

int main()
{
    S s = g();
    cout << s.name << " " << s.num << endl;
    return 0;
}

```

2. Retornar um objeto std::tupla ou std::p Air:

```

#include <tuple>
#include <string>
#include <iostream>

using namespace std;

tuple<int, string, double> f()
{
    int i{ 108 };
    string s{ "Some text" };
    double d{ .01 };
    return { i,s,d };
}

int main()
{
    auto t = f();
    cout << get<0>(t) << " " << get<1>(t) << " " << get<2>(t) << endl;

    // --or--

    int myval;
    string myname;
    double mydecimal;
    tie(myval, myname, mydecimal) = f();
    cout << myval << " " << myname << " " << mydecimal << endl;

    return 0;
}

```

3. **Visual Studio 2017 versão 15,3 e posterior** (disponível com `/std:c++17`): usar associações estruturadas. A vantagem das associações estruturadas é que as variáveis que armazenam os valores de retorno são inicializadas ao mesmo tempo em que são declaradas, o que, em alguns casos, pode ser significativamente mais eficiente. Na instrução `auto[x, y, z] = f();` , os colchetes apresentam e inicializam os nomes que estão no escopo do bloco de função inteiro.

```

#include <tuple>
#include <string>
#include <iostream>

using namespace std;

tuple<int, string, double> f()
{
    int i{ 108 };
    string s{ "Some text" };
    double d{ .01 };
    return { i, s, d };
}

struct S
{
    string name;
    int num;
};

S g()
{
    string t{ "hello" };
    int u{ 42 };
    return { t, u };
}

int main()
{
    auto[x, y, z] = f(); // init from tuple
    cout << x << " " << y << " " << z << endl;

    auto[a, b] = g(); // init from POD struct
    cout << a << " " << b << endl;
    return 0;
}

```

4. Além de usar o próprio valor de retorno, você pode "retornar" valores definindo qualquer número de parâmetros para usar passagem por referência para que a função possa modificar ou inicializar os valores dos objetos que o chamador fornece. Para obter mais informações, consulte [argumentos da função de tipo de referência](#).

Ponteiros de função

O C++ dá suporte a ponteiros de função da mesma maneira que a linguagem C. No entanto, uma alternativa mais segura de tipo é geralmente usar um objeto de função.

É recomendável que `typedef` seja usado para declarar um alias para o tipo de ponteiro de função se declarar uma função que retorne um tipo de ponteiro de função. Por exemplo

```

typedef int (*fp)(int);
fp myFunction(char* s); // function returning function pointer

```

Se isso não for feito, a sintaxe adequada para a declaração de função pode ser deduzida da sintaxe do declarador para o ponteiro de função substituindo o identificador (`fp` no exemplo acima) pelo nome das funções e a lista de argumentos, como segue:

```

int (*myFunction(char* s))(int);

```

A declaração anterior é equivalente à declaração usando `typedef` acima.

Confira também

[Sobrecarga de função](#)

[Funções com listas de argumentos variáveis](#)

[Funções explicitamente padronizadas e excluídas](#)

[Consulta de nome dependente de argumento \(Koenig\) em funções](#)

[Argumentos padrão](#)

[Funções embutidas](#)

Funções com listas de argumentos variáveis (C++)

02/09/2020 • 5 minutes to read • [Edit Online](#)

Declarações de função em que o último membro de é reticências (...) podem usar um número variável de argumentos. Nesses casos, o C++ fornece verificação de tipo apenas para os argumentos explicitamente declarados. Você pode usar listas de argumentos variáveis quando precisar criar uma função tão geral que mesmo o número e os tipos de argumentos possam variar. A família de funções é um exemplo de funções que usam listas de argumentos variáveis. `printf` *declaração de argumento-lista*

Funções com argumentos variáveis

Para acessar argumentos depois daqueles declarados, use as macros contidas no arquivo de inclusão padrão, `<stdarg.h>` conforme descrito abaixo.

Específico da Microsoft

O Microsoft C++ permite que as reticências sejam especificadas como um argumento se as reticências forem o último argumento e se forem precedidas por uma vírgula. Consequentemente, a declaração

`int Func(int i, ...);` é válida, mas `int Func(int i ...);` não é.

FINAL específico da Microsoft

A declaração de uma função que pega um número variável de argumentos que requer pelo menos um argumento de espaço reservado, mesmo se não for usado. Se esse argumento de espaço reservado não for fornecido, não há como acessar os argumentos restantes.

Quando argumentos do tipo `char` são passados como argumentos variáveis, eles são convertidos em tipo `int`. Da mesma forma, quando argumentos do tipo `float` são passados como argumentos variáveis, eles são convertidos em tipo `double`. Os argumentos de outros tipos estão sujeitos às promoções integral e de ponto flutuante comuns. Consulte [conversões padrão](#) para obter mais informações.

As funções que exigem listas de variáveis são declaradas usando as reticências (...) na lista de argumentos. Use os tipos e macros descritos no `<stdarg.h>` arquivo include para acessar os argumentos que são passados por uma lista de variáveis. Para obter mais informações sobre essas macros, consulte [va_arg](#), [va_copy](#), [va_end](#) [va_start](#) na documentação da biblioteca de tempo de execução do C.

O exemplo a seguir mostra como as macros funcionam em conjunto com o tipo (declarado em `<stdarg.h>`):

```
// variable_argument_lists.cpp
#include <stdio.h>
#include <stdarg.h>

// Declaration, but not definition, of ShowVar.
void ShowVar( char *szTypes, ... );
int main() {
    ShowVar( "fcsi", 32.4f, 'a', "Test string", 4 );
}

// ShowVar takes a format string of the form
// "ifcs", where each character specifies the
// type of the argument in that position.
//
// i = int
// f = float
// c = char
// s = string (char *)
```

```

// Following the format specification is a variable
// list of arguments. Each argument corresponds to
// a format character in the format string to which
// the szTypes parameter points
void ShowVar( char *szTypes, ... ) {
    va_list vl;
    int i;

    // szTypes is the last argument specified; you must access
    // all others using the variable-argument macros.
    va_start( vl, szTypes );

    // Step through the list.
    for( i = 0; szTypes[i] != '\0'; ++i ) {
        union Printable_t {
            int      i;
            float    f;
            char     c;
            char    *s;
        } Printable;
        switch( szTypes[i] ) {    // Type to expect.
            case 'i':
                Printable.i = va_arg( vl, int );
                printf_s( "%i\n", Printable.i );
                break;

            case 'f':
                Printable.f = va_arg( vl, double );
                printf_s( "%f\n", Printable.f );
                break;

            case 'c':
                Printable.c = va_arg( vl, char );
                printf_s( "%c\n", Printable.c );
                break;

            case 's':
                Printable.s = va_arg( vl, char * );
                printf_s( "%s\n", Printable.s );
                break;

            default:
                break;
        }
    }
    va_end( vl );
}

//Output:
// 32.400002
// a
// Test string

```

O exemplo anterior ilustra estes conceitos importantes:

1. Você deve estabelecer um marcador de lista como uma variável de tipo `va_list` antes que argumentos de variáveis sejam acessados. No exemplo anterior, o marcador é chamado `vl`.
2. Os argumentos individuais são acessados usando a macro `va_arg`. Você deve informar à macro `va_arg` o tipo de argumento a ser recuperado para que ela possa transferir o número de bytes correto da pilha. Se você especificar um tipo incorreto de um tamanho diferente do fornecido pelo programa de chamada para `va_arg`, os resultados serão imprevisíveis.
3. É necessário converter explicitamente o resultado obtido usando a macro `va_arg` no tipo desejado.

Você deve chamar a macro para encerrar o processamento de argumento de variáveis. `va_end`

Sobrecarga de função

02/09/2020 • 34 minutes to read • [Edit Online](#)

O C++ permite a especificação de mais de uma função do mesmo nome no mesmo escopo. Essas funções são chamadas de funções *sobre carregadas*. Funções sobre carregadas permitem que você forneça semânticas diferentes para uma função, dependendo dos tipos e do número de argumentos.

Por exemplo, uma `print` função que usa um `std::string` argumento pode executar tarefas muito diferentes de uma que usa um argumento do tipo `double`. A sobre carregada evita que você precise usar nomes como `print_string` ou `print_double`. No momento da compilação, o compilador escolhe qual sobre carregada usar com base no tipo de argumentos passado pelo chamador. Se você chamar `print(42.0)`, a `void print(double d)` função será invocada. Se você chamar `print("hello world")`, a `void print(std::string)` sobre carregada será invocada.

Você pode sobre carregar ambas as funções membro e não membro. A tabela a seguir mostra quais partes de uma declaração de função C++ usa para diferenciar entre grupos de funções com o mesmo nome no mesmo escopo.

Considerações de sobre carregada

ELEMENTO DE DECLARAÇÃO DE FUNÇÃO	USADO PARA SOBRECARGA?
Tipo de retorno de função	Não
Número de argumentos	Sim
Tipo de argumentos	Sim
Presença ou ausência de reticências	Sim
Uso de <code>typedef</code> nomes	Não
Limites de matriz não especificados	Não
<code>const</code> or** <code>volatile</code>	Sim, quando aplicado à função inteira
Qualificadores de referência	Sim

Exemplo

O exemplo a seguir ilustra como a sobre carregada pode ser usada.

```
// function_overloading.cpp
// compile with: /EHsc
#include <iostream>
#include <math.h>
#include <string>

// Prototype three print functions.
int print(std::string s);           // Print a string.
int print(double dvalue);          // Print a double.
int print(double dvalue, int prec); // Print a double with a
```

```

        // given precision.

using namespace std;
int main(int argc, char *argv[])
{
    const double d = 893094.2987;
    if (argc < 2)
    {
        // These calls to print invoke print( char *s ).
        print("This program requires one argument.");
        print("The argument specifies the number of");
        print("digits precision for the second number");
        print("printed.");
        exit(0);
    }

    // Invoke print( double dvalue ).
    print(d);

    // Invoke print( double dvalue, int prec ).
    print(d, atoi(argv[1]));
}

// Print a string.
int print(string s)
{
    cout << s << endl;
    return cout.good();
}

// Print a double in default precision.
int print(double dvalue)
{
    cout << dvalue << endl;
    return cout.good();
}

// Print a double in specified precision.
// Positive numbers for precision indicate how many digits
// precision after the decimal point to show. Negative
// numbers for precision indicate where to round the number
// to the left of the decimal point.
int print(double dvalue, int prec)
{
    // Use table-lookup for rounding/truncation.
    static const double rgPow10[] = {
        10E-7, 10E-6, 10E-5, 10E-4, 10E-3, 10E-2, 10E-1,
        10E0, 10E1, 10E2, 10E3, 10E4, 10E5, 10E6 };
    const int iPowZero = 6;

    // If precision out of range, just print the number.
    if (prec < -6 || prec > 7)
    {
        return print(dvalue);
    }
    // Scale, truncate, then rescale.
    dvalue = floor(dvalue / rgPow10[iPowZero - prec]) *
        rgPow10[iPowZero - prec];
    cout << dvalue << endl;
    return cout.good();
}

```

O código anterior mostra a sobreposição da função `print` no escopo do arquivo.

O argumento padrão não é considerado parte do tipo de função. Portanto, ele não é usado na seleção de funções sobreporadas. Duas funções que diferem nos argumentos padrão são consideradas várias definições em vez de funções sobreporadas.

Os argumentos padrão não podem ser fornecidos para operadores sobrecarregados.

Correspondência de argumento

As funções sobrecarregadas são selecionadas para corresponder melhor declarações de função no escopo atual aos argumentos fornecidos na chamada de função. Se uma função apropriada for localizada, essa função é chamada. "Adequado" neste contexto significa:

- Uma correspondência exata foi encontrada.
- Uma conversão trivial foi executada.
- Uma promoção integral foi executada.
- Uma conversão padrão para o tipo de argumento desejado existe.
- Uma conversão definida pelo usuário (operador ou construtor de conversão) para o tipo de argumento desejado existe.
- Argumentos representados por reticências foram encontrados.

O compilador cria um conjunto de funções candidatas para cada argumento. As funções candidatas são funções em que o argumento real nessa posição pode ser convertido no tipo do argumento formal.

Um conjunto de "melhores funções correspondentes" é criado para cada argumento, e a função selecionada é a interseção de todos os conjuntos. Se a interseção contiver mais de uma função, a sobrecarga é ambígua e gera um erro. A função que é selecionada sempre é uma correspondência melhor de que todas as outras funções no grupo para no mínimo um argumento. Se não houver um vencedor claro, a chamada de função gerará um erro.

Observe as seguintes declarações (as funções são marcadas `Variant 1`, `Variant 2` e `Variant 3`, para identificação na discussão a seguir):

```
Fraction &Add( Fraction &f, long l );           // Variant 1
Fraction &Add( long l, Fraction &f );           // Variant 2
Fraction &Add( Fraction &f, Fraction &f );       // Variant 3

Fraction F1, F2;
```

Considere o seguinte instrução:

```
F1 = Add( F2, 23 );
```

A instrução anterior compila dois conjuntos:

CONJUNTO 1: FUNÇÕES CANDIDATAS QUE TÊM O PRIMEIRO ARGUMENTO DO TIPO FRAÇÃO	SET 2: FUNÇÕES CANDIDATAS CUJO SEGUNDO ARGUMENTO PODE SER CONVERTIDO EM TIPO** <code>INT</code> **
Variant 1	Variante 1 (<code>int</code> pode ser convertido para <code>long</code> usando uma conversão padrão)
Variante 3	

As funções no conjunto 2 são funções para as quais há conversões implícitas do tipo de parâmetro real para o tipo de parâmetro formal e entre essas funções há uma função para a qual o "custo" de converter o tipo de parâmetro real para seu tipo de parâmetro formal é o menor.

A interseção desses dois conjuntos é a Variant 1. Um exemplo de uma chamada de função ambígua é:

```
F1 = Add( 3, 6 );
```

A chamada de função anterior compila os seguintes conjuntos:

SET 1: FUNÇÕES CANDIDATAS QUE TÊM O PRIMEIRO ARGUMENTO DO TIPO** `int` **

Variante 2 (`int` pode ser convertido para `long` usando uma conversão padrão)

SET 2: FUNÇÕES CANDIDATAS QUE TÊM O SEGUNDO ARGUMENTO DO TIPO** `int` **

Variante 1 (`int` pode ser convertido para `long` usando uma conversão padrão)

Como a interseção desses dois conjuntos está vazia, o compilador gera uma mensagem de erro.

Para correspondência de argumentos, uma função com n argumentos padrão é tratada como $n+1$ funções separadas, cada uma com um número diferente de argumentos.

As reticências (...) atuam como um curinga; elas correspondem a qualquer argumento real. Pode levar a muitos conjuntos ambíguos, se você não projetar seus conjuntos de funções sobrecarregados com extrema atenção.

NOTE

A ambiguidade de funções sobrecarregadas não pode ser determinada até que uma chamada de função seja encontrada. Nesse ponto, os conjuntos são compilados para cada argumento na chamada de função, e você pode determinar se há uma sobrecarga inequívoca. Isso significa que as ambiguidades podem permanecer em seu código até que sejam evocadas por uma chamada de função específica.

Diferenças de tipo de argumento

As funções sobrecarregadas diferenciam-se entre os tipos de argumento que têm inicializadores diferentes. Portanto, um argumento de um determinado tipo e uma referência a esse tipo são considerados iguais para fins de sobrecarga. Eles são considerados iguais porque têm os mesmos inicializadores. Por exemplo, `max(double, double)` é considerado o mesmo que `max(double &, double &)`. Declarar essas duas funções causa um erro.

Pelo mesmo motivo, os argumentos de função de um tipo modificado por `const` ou `volatile` não são tratados de forma diferente do tipo base para fins de sobrecarga.

No entanto, o mecanismo de sobrecarga de função pode distinguir entre as referências que são qualificadas por `const` e `volatile` e referências ao tipo base. Ele torna o código como o seguinte possível:

```

// argument_type_differences.cpp
// compile with: /EHsc /W3
// C4521 expected
#include <iostream>

using namespace std;
class Over {
public:
    Over() { cout << "Over default constructor\n"; }
    Over( Over &o ) { cout << "Over&\n"; }
    Over( const Over &o ) { cout << "const Over&\n"; }
    Over( volatile Over &o ) { cout << "volatile Over&\n"; }
};

int main() {
    Over o1;           // Calls default constructor.
    Over o2( o1 );    // Calls Over( Over& ). 
    const Over o3;    // Calls default constructor.
    Over o4( o3 );    // Calls Over( const Over& ). 
    volatile Over o5; // Calls default constructor.
    Over o6( o5 );    // Calls Over( volatile Over& ). 
}

```

Saída

```

Over default constructor
Over&
Over default constructor
const Over&
Over default constructor
volatile Over&

```

Ponteiros para `const` e `volatile` objetos também são considerados diferentes de ponteiros para o tipo base para fins de sobrecarga.

Correspondência de argumentos e conversões

Quando o compilador tenta corresponder argumentos reais com os argumentos em declarações de função, ele pode fornecer conversões padrão ou definidas pelo usuário para obter o tipo correto se nenhuma correspondência exata for encontrada. A aplicação de conversões está sujeita a estas regras:

- As sequências de conversões que contêm mais de uma conversão definida pelo usuário não são consideradas.
- As sequências de conversões que podem ser encurtadas removendo as conversões intermediárias não são consideradas.

A sequência de conversões resultante, se houver, será considerada a melhor sequência de correspondência. Há várias maneiras de converter um objeto do tipo `int` para o tipo `unsigned long` usando conversões padrão (descritas em [conversões padrão](#)):

- Converta de `int` para `long` e, em seguida, de `long` para `unsigned long`.
- Converter de `int` para `unsigned long`.

A primeira sequência, embora alcance a meta desejada, não é a melhor sequência de correspondência — existe uma sequência mais curta.

A tabela a seguir mostra um grupo de conversões, as conversões triviais chamadas, que têm um efeito limitado na determinação de qual sequência é a melhor correspondência. As instâncias em que as conversões triviais

afetam a escolha de sequência são abordadas na lista após a tabela.

Conversões triviais

CONVERTER DO TIPO	CONVERTER NO TIPO
<i>nome do tipo</i>	<i>nome do tipo</i> ***&*
<i>nome do tipo</i> ***&*	<i>nome do tipo</i>
<i>nome-do-tipo []</i>	<i>nome do tipo</i> *
<i>nome-tipo (lista de argumentos)</i>	(* <i>nome-do-tipo</i>) (<i>lista de argumentos</i>)
<i>nome do tipo</i>	* <code>const</code> *** <i>nome do tipo</i>
<i>nome do tipo</i>	* <code>volatile</code> *** <i>nome do tipo</i>
<i>nome do tipo</i> *	* <code>const</code> *** <i>nome do tipo</i> *
<i>nome do tipo</i> *	* <code>volatile</code> *** <i>nome do tipo</i> *

A sequência em que as conversões são executadas é a seguinte:

1. Correspondência exata. Uma correspondência exata entre os tipos com que a função é chamada e os tipos declarados no protótipo da função sempre é a melhor correspondência. As sequências de conversões triviais são classificadas como correspondências exatas. No entanto, as sequências que não fazem nenhuma dessas conversões são consideradas melhores do que as sequências que são convertidas:
 - De ponteiro, para ponteiro para `const` (`type` * para `const` `type` *).
 - De ponteiro, para ponteiro para `volatile` (`type` * para `volatile` `type` *).
 - De referência, para fazer referência a `const` (`type` & para `const` `type` &).
 - De referência, para fazer referência a `volatile` (`type` & para `volatile` `type` &).
2. Correspondência usando promoções. Qualquer sequência não classificada como uma correspondência exata que contenha apenas promoções integrais, conversões de `float` para `double` e conversões triviais é classificada como uma correspondência usando promoções. Embora não seja tão boa quanto a correspondência exata, a correspondência usando promoções é melhor do que a correspondência usando conversões padrão.
3. Correspondência usando conversões padrão. Qualquer sequência não classificada como correspondência exata ou correspondência usando promoções que contém somente conversões padrão e conversões triviais é classificada como correspondência usando conversões padrão. Nessa categoria, as seguintes regras são aplicadas:
 - A conversão de um ponteiro para uma classe derivada, para um ponteiro para uma classe base direta ou indireta, é preferível para converter para `void *` ou `const void *`.
 - A conversão de um ponteiro em uma classe derivada, em um ponteiro em uma classe base gera uma correspondência melhor quanto mais próxima a classe base estiver de uma classe base direta. Suponhamos que a hierarquia de classes seja a ilustrada na figura a seguir.



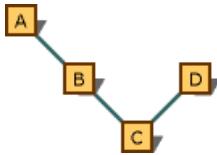
Gráfico mostrando conversões preferenciais

A conversão do tipo D^* no tipo C^* é preferível à conversão do tipo D^* no tipo B^* . Da mesma forma, a conversão do tipo D^* no tipo B^* é preferível à conversão do tipo D^* no tipo A^* .

Essa mesma regra se aplica para referenciar conversões. A conversão do tipo $D&$ no tipo $C&$ é preferível à conversão do tipo $D&$ no tipo $B&$.

Essa mesma regra se aplica às conversões de ponteiro em membro. A conversão do tipo $T D::^*$ no tipo $T C::^*$ é preferível à conversão do tipo $T D::^*$ no tipo $T B::^*$, e assim por diante (onde T é o tipo do membro).

A regra anterior só se aplica ao longo de um caminho específico de derivação. Considere o gráfico mostrado na figura a seguir.



Grafo de várias heranças que mostra conversões preferenciais

A conversão do tipo C^* no tipo B^* é preferível à conversão do tipo C^* no tipo A^* . A razão é que eles estão no mesmo caminho, e B^* é mais próximo. No entanto, a conversão de tipo C^* para tipo D^* não é preferível à conversão para tipo A^* ; não há preferência porque as conversões seguem caminhos diferentes.

1. Correspondência com conversões definidas pelo usuário. Essa sequência não pode ser classificada como uma correspondência exata, uma correspondência usando promoções ou uma correspondência usando conversões padrão. A sequência deve conter apenas conversões definidas pelo usuário, conversões padrão ou conversões triviais para ser classificada como correspondência com conversões definidas pelo usuário. Uma correspondência com conversões definidas pelo usuário é considerada uma correspondência melhor do que uma correspondência com um sinal de reticências, mas tão boa quanto uma correspondência com conversões padrão.
2. Correspondência com um sinal de reticências. Qualquer sequência que corresponda a reticências na declaração é classificada como correspondência com um sinal de reticências. Ela é considerada a correspondência mais fraca.

As conversões definidas pelo usuário são aplicadas quando não há promoção ou conversão. Essas conversões são selecionadas com base no tipo do argumento que está sendo correspondido. Considere o seguinte código:

```

// argument_matching1.cpp
class UDC
{
public:
    operator int()
    {
        return 0;
    }
    operator long();
};

void Print( int i )
{
};

UDC udc;

int main()
{
    Print( udc );
}

```

As conversões disponíveis definidas pelo usuário para a classe `UDC` são do tipo `int` e do tipo `long`. Portanto, o compilador considera conversões para o tipo do objeto que está sendo correspondido: `UDC`. Uma conversão para `int` existe e está selecionada.

Durante o processo de correspondência de argumentos, as conversões padrão podem ser aplicadas ao argumento e ao resultado de uma conversão definida pelo usuário. Portanto, o código a seguir funciona:

```

void LogToFile( long l );
...
UDC udc;
LogToFile( udc );

```

No exemplo anterior, a conversão definida pelo usuário, **operador Long**, é invocada para converter `udc` em Type `long`. Se nenhuma conversão definida pelo usuário para o tipo `long` tivesse sido definida, a conversão teria prosseguido da seguinte maneira: o tipo `UDC` teria sido convertido para `int` o tipo usando a conversão definida pelo usuário. Em seguida, a conversão padrão de tipo `int` para tipo `long` teria sido aplicada para corresponder ao argumento na declaração.

Se qualquer conversões definida pelo usuário for necessária para corresponder a um argumento, as conversões padrão não serão usadas ao avaliar a melhor correspondência. Mesmo que mais de uma função candidata exija uma conversão definida pelo usuário, as funções serão consideradas iguais. Por exemplo:

```

// argument_matching2.cpp
// C2668 expected
class UDC1
{
public:
    UDC1( int ); // User-defined conversion from int.
};

class UDC2
{
public:
    UDC2( long ); // User-defined conversion from long.
};

void Func( UDC1 );
void Func( UDC2 );

int main()
{
    Func( 1 );
}

```

Ambas as versões do `Func` exigem uma conversão definida pelo usuário para converter `int` o tipo para o argumento de tipo de classe. As conversões possíveis são:

- Converter de tipo `int` para tipo `UDC1` (uma conversão definida pelo usuário).
- Converter de tipo `int` para tipo `long`; em seguida, converter em tipo `UDC2` (uma conversão em duas etapas).

Embora o segundo exija uma conversão padrão e a conversão definida pelo usuário, as duas conversões ainda são consideradas iguais.

NOTE

As conversões definidas pelo usuário são consideradas conversão por construção ou conversão por inicialização (função de conversão). Ambos os métodos são considerados iguais ao considerar a melhor correspondência.

Correspondência de argumentos e o ponteiro

As funções de membro de classe são tratadas de forma diferente, dependendo se elas são declaradas como `static`. Como as funções não estáticas têm um argumento implícito que fornece o `this` ponteiro, as funções não estáticas são consideradas para ter mais um argumento do que funções estáticas; caso contrário, elas são declaradas de forma idêntica.

Essas funções de membro não estáticos exigem que o `this` ponteiro implícito corresponda ao tipo de objeto por meio do qual a função está sendo chamada ou, para operadores sobrearcarregados, eles exigem que o primeiro argumento corresponda ao objeto no qual o operador está sendo aplicado. (Para obter mais informações sobre operadores sobrearcarregados, consulte [operadores sobrearcarregados](#).)

Ao contrário de outros argumentos em funções sobrearcarregadas, nenhum objeto temporário é introduzido e nenhuma tentativa de conversões é feita ao tentar corresponder o `this` argumento de ponteiro.

Quando o `->` operador de seleção de membro é usado para acessar uma função de membro da classe `class_name`, o `this` argumento de ponteiro tem um tipo de `class_name * const`. Se os membros forem declarados como `const` ou `volatile`, os tipos serão `const class_name * const` e `volatile class_name * const`, respectivamente.

O operador de seleção de membro `.` funciona exatamente da mesma maneira, exceto que um operador `&` (address-of) implícito tem um prefixo no nome do objeto. O exemplo a seguir mostra como isso funciona:

```
// Expression encountered in code
obj.name

// How the compiler treats it
(&obj)->name
```

O operando esquerdo dos `->*` operadores e `.*` (ponteiro para membro) são tratados da mesma forma que os `.` operadores e `->` (seleção de membro) em relação à correspondência de argumentos.

Qualificadores de referência em funções de membro

Os qualificadores de referência possibilitam a sobrecarga de uma função de membro com base em se o objeto apontado pelo `this` é um Rvalue ou um lvalue. Esse recurso pode ser usado para evitar operações de cópia desnecessárias em cenários em que você opta por não fornecer acesso de ponteiro aos dados. Por exemplo, suponha `C` que classe inicialize alguns dados em seu construtor e retorne uma cópia desses dados na função de membro `get_data()`. Se um objeto do tipo `C` for um Rvalue que está prestes a ser destruído, o compilador escolherá a `get_data() &&` sobrecarga, que moverá os dados em vez de copiá-los.

```
#include <iostream>
#include <vector>

using namespace std;

class C
{

public:
    C() /*expensive initialization*/
    vector<unsigned> get_data() &
    {
        cout << "lvalue\n";
        return _data;
    }
    vector<unsigned> get_data() &&
    {
        cout << "rvalue\n";
        return std::move(_data);
    }

private:
    vector<unsigned> _data;
};

int main()
{
    C c;
    auto v = c.get_data(); // get a copy. prints "lvalue".
    auto v2 = C().get_data(); // get the original. prints "rvalue"
    return 0;
}
```

Restrições sobre sobrecarga

Várias restrições regem um conjunto de funções sobrecarregadas aceitável:

- Quaisquer duas funções em um conjunto de funções sobrecarregadas devem ter listas de argumentos diferentes.

- Funções sobrecarregadas com listas de argumentos dos mesmos tipos, com base apenas no tipo de retorno, são um erro.

Específico da Microsoft

Você pode sobrepor o **operador novo** exclusivamente na base do tipo de retorno — especificamente, com base no modificador de modelo de memória especificado.

FINAL específico da Microsoft

- As funções de membro não podem ser sobrepor exclusivamente com base em uma que seja estática e outra não estática.
- `typedef` as declarações não definem novos tipos; Eles apresentam sinônimos para tipos existentes. Eles não afetam o mecanismo de sobrepor. Considere o seguinte código:

```
typedef char * PSTR;

void Print( char *szToPrint );
void Print( PSTR szToPrint );
```

As duas funções anteriores têm listas de argumento idênticas. `PSTR` é um sinônimo para o tipo `char *`. No escopo do membro, esse código gera um erro.

- Os tipos enumerados são tipos distintos e podem ser usados para distinguir as funções sobrepor.
- Os tipos "matriz de" e "ponteiro para" são considerados idênticos para fins de distinção entre funções sobrepor, mas somente para matrizes com dimensão única. É por isso que essas funções sobrepor entram em conflito e geram uma mensagem de erro:

```
void Print( char *szToPrint );
void Print( char szToPrint[] );
```

Para matrizes dimensionadas multiplamente, a segunda e todas as dimensões seguintes são consideradas parte do tipo. Portanto, elas são usadas na distinção das funções sobrepor:

```
void Print( char szToPrint[] );
void Print( char szToPrint[][7] );
void Print( char szToPrint[][9][42] );
```

Sobrepor, substituindo e ocultando

Quaisquer duas declarações de função do mesmo nome no mesmo escopo podem fazer referência à mesma função, ou duas funções discretas sobrepor. Se as listas de argumento de declarações contiverem argumentos de tipos equivalentes (como descrito na seção anterior), as declarações de função se referem à mesma função. Se não, fazem referência a duas funções diferentes que são selecionadas usando a sobrepor.

O escopo da classe é estritamente observado; Portanto, uma função declarada em uma classe base não está no mesmo escopo que uma função declarada em uma classe derivada. Se uma função em uma classe derivada for declarada com o mesmo nome de uma função virtual na classe base, a função de classe derivada *substituirá* a função de classe base. Para obter mais informações, consulte [funções virtuais](#).

Se a função de classe base não for declarada como ' virtual ', a função de classe derivada será indicada para *ocultá-la*. Tanto a substituição quanto a ocultação são diferentes da sobrepor.

O escopo do bloco é estritamente observado; Portanto, uma função declarada no escopo do arquivo não está no mesmo escopo que uma função declarada localmente. Se uma função declarada localmente tiver o mesmo nome de uma função declarada no escopo de arquivo, a função declarada localmente oculta a função do escopo de arquivo ao invés de causar a sobrecarga. Por exemplo:

```
// declaration_matching1.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
void func( int i )
{
    cout << "Called file-scoped func : " << i << endl;
}

void func( char *sz )
{
    cout << "Called locally declared func : " << sz << endl;
}

int main()
{
    // Declare func local to main.
    extern void func( char *sz );

    func( 3 );    // C2664 Error. func( int ) is hidden.
    func( "s" );
}
```

O código anterior mostra duas definições da função `func`. A definição que usa um argumento do tipo `char *` é local para `main` por causa da `extern` instrução. Portanto, a definição que usa um argumento do tipo `int` é ocultada e a primeira chamada para `func` está em erro.

Para funções de membro sobrecarregadas, as versões diferentes da função podem receber privilégios de acesso diferentes. Elas são consideradas como ainda no escopo da classe envolvente e, portanto, são funções sobrecarregadas. Considere o seguinte código, no qual a função de membro `Deposit` é sobrecarregada; uma versão é pública, a outro, privada.

A finalidade deste exemplo é fornecer uma classe `Account` em que uma senha correta é necessária para executar depósitos. Isso é feito com o sobreescrita.

A chamada para `Deposit` em `Account::Deposit` chama a função de membro particular. Essa chamada está correta porque `Account::Deposit` é uma função membro e tem acesso aos membros privados da classe.

```

// declaration_matching2.cpp
class Account
{
public:
    Account()
    {
    }
    double Deposit( double dAmount, char *szPassword );
private:
    double Deposit( double dAmount )
    {
        return 0.0;
    }
    int Validate( char *szPassword )
    {
        return 0;
    }
};

int main()
{
    // Allocate a new object of type Account.
    Account *pAcct = new Account;

    // Deposit $57.22. Error: calls a private function.
    // pAcct->Deposit( 57.22 );

    // Deposit $57.22 and supply a password. OK: calls a
    // public function.
    pAcct->Deposit( 52.77, "pswd" );
}

double Account::Deposit( double dAmount, char *szPassword )
{
    if ( Validate( szPassword ) )
        return Deposit( dAmount );
    else
        return 0.0;
}

```

Confira também

[Funções \(C++\)](#)

Funções explicitamente usadas como padrão e excluídas

02/09/2020 • 12 minutes to read • [Edit Online](#)

No C++11, as funções usadas como padrão e excluídas proporcionam controle explícito sobre se as funções de membro especial serão geradas automaticamente. As funções excluídas também oferecem uma linguagem simples para impedir que promoções de tipos problemáticos ocorram em argumentos para funções de todos os tipos — funções de membro especial, funções de membro normal e funções de não membro — que, de outra forma, causariam uma chamada de função indesejada.

Benefícios das funções explicitamente usadas como padrão e excluídas

No C++, o compilador gera automaticamente o construtor padrão, o construtor de cópia, o operador de atribuição de cópia e o destruidor para um tipo se ele não declarar os próprios. Essas funções são conhecidas como *funções de membro especiais*, e são o que fazem com que tipos simples definidos pelo usuário em C++ se comportem como estruturas no C. Ou seja, você pode criar, copiar e destruí-los sem qualquer esforço de codificação adicional. O C++11 traz a semântica de movimentação para a linguagem e adiciona o construtor de movimentação e o operador de atribuição de movimentação à lista de funções de membro especial que o compilador pode gerar automaticamente.

Isso é conveniente para tipos simples, mas os tipos complexos geralmente definem, por conta própria, uma ou mais das funções de membro especial, e isso pode impedir que outras funções de membro especial sejam geradas automaticamente. Na prática:

- Se qualquer construtor for declarado explicitamente, nenhum construtor padrão será gerado automaticamente.
- Se um destruidor virtual for declarado explicitamente, nenhum destruidor padrão será gerado automaticamente.
- Se um construtor de movimentação ou um operador de atribuição de movimentação for declarado explicitamente:
 - Nenhum construtor de cópia será gerado automaticamente.
 - Nenhum operador de atribuição de cópia será gerado automaticamente.
- Se um construtor de cópia, um operador de atribuição de cópia, um construtor de movimentação, um operador de atribuição de movimentação ou um destruidor for declarado explicitamente:
 - Nenhum construtor de movimentação será gerado automaticamente.
 - Nenhum operador de atribuição de movimentação será gerado automaticamente.

NOTE

Além disso, o padrão C++11 especifica as seguintes regras adicionais:

- Se um construtor de cópia ou um destruidor for declarado explicitamente, a geração automática do operador de atribuição de cópia será preterida.
- Se um operador de atribuição de cópia ou um destruidor for declarado explicitamente, a geração automática do construtor de cópia será preterida.

Nos dois casos, o Visual Studio continua a gerar as funções necessárias de forma automática e implícita, e não emite um aviso.

As consequências dessas regras também podem vazar para as hierarquias de objetos. Por exemplo, se por algum motivo uma classe base não tiver um construtor padrão que possa ser chamado por meio de uma classe de derivação — ou seja, um `public` `protected` Construtor ou que não usa parâmetros — então, uma classe derivada dela não poderá gerar automaticamente seu próprio construtor padrão.

Essas regras podem complicar a implementação do que deveria ser tipos simples definidos pelo usuário e expressões comuns de C++; por exemplo, podem tornar um tipo definido pelo usuário não copiável declarando o construtor de cópia e o operador de atribuição de cópia de forma privada, sem defini-los.

```
struct noncopyable
{
    noncopyable() {};

private:
    noncopyable(const noncopyable&);
    noncopyable& operator=(const noncopyable&);
};
```

Antes do C++11, esse snippet de código era a forma idiomática de tipos não copiáveis. No entanto, existem vários problemas:

- O construtor de cópia deve ser declarado de forma privada para ocultá-lo, mas, como ele está declarado, a geração automática do construtor padrão é impedida. Você terá que definir o construtor padrão explicitamente se quiser um, mesmo que ele não faça nada.
- Mesmo se o construtor padrão definido explicitamente não fizer nada, será considerado não trivial pelo compilador. É menos eficiente do que um construtor padrão gerado automaticamente e impede que `noncopyable` seja um tipo POD verdadeiro.
- Mesmo que o construtor de cópia e o operador de atribuição de cópia estejam ocultos do código externo, as funções de membro e os amigos de `noncopyable` ainda poderão vê-los e chamá-los. Se eles forem declarados, mas não definidos, chamá-los causará um erro de vinculador.
- Embora essa seja uma expressão aceita de modo geral, a intenção não fica clara, a menos que você compreenda todas as regras para a geração automática das funções de membro especial.

No C++11, a expressão não copiável pode ser implementada de maneira mais simples.

```
struct noncopyable
{
    noncopyable() =default;
    noncopyable(const noncopyable&) =delete;
    noncopyable& operator=(const noncopyable&) =delete;
};
```

Observe como os problemas com a expressão pré-C++11 são resolvidos:

- A geração do construtor padrão ainda é impedida por meio da declaração do construtor de cópia, mas você pode trazê-lo de volta ao usá-lo como padrão explicitamente.
- As funções de membro especial explicitamente usadas como padrão ainda são consideradas triviais; portanto, não há penalidade de desempenho, e `noncopyable` não é impedido de ser um tipo POD verdadeiro.
- O construtor de cópia e o operador de atribuição de cópia são públicos, mas são excluídos. É um erro em tempo de compilação definir ou chamar uma função excluída.
- A intenção é clara para qualquer pessoa que compreenda `=default` e `=delete`. Você não precisa compreender as regras para a geração automática de funções de membro especial.

Existem expressões semelhantes para criar tipos definidos pelo usuário que não podem ser movidos, que só podem ser alocados dinamicamente ou que não podem ser alocados dinamicamente. Cada uma dessas expressões tem implementações pré-C++11 que sofrem problemas semelhantes, e que são resolvidas de forma semelhante no C++11 pela implementação delas em termos de funções de membro especial usadas como padrão e excluídas.

Funções explicitamente usadas como padrão

Você pode usar como padrão qualquer uma das funções de membro especial: para declarar explicitamente que a função de membro especial usa a implementação padrão, para definir a função de membro especial com um qualificador de acesso não público ou para declarar novamente uma função de membro especial cuja geração automática foi impedida por outras circunstâncias.

Para usar uma função de membro especial como padrão, você a declara, como neste exemplo:

```
struct widget
{
    widget()=default;

    inline widget& operator=(const widget&);
};

inline widget& widget::operator=(const widget&) =default;
```

Observe que você pode padronizar uma função de membro especial fora do corpo de uma classe desde que ela seja inlinable.

Por causa dos benefícios de desempenho das funções de membro especial triviais, recomendamos que você prefira as funções de membro especial geradas automaticamente aos corpos de função vazios quando quiser o comportamento padrão. Você pode fazer isso usando a função de membro especial como padrão explicitamente, ou não a declarando (e também não declarando outras funções de membro especial que a impediriam de ser gerada automaticamente).

Funções excluídas

Você pode excluir funções de membro especial, assim como funções de membro normal e funções de não membro, para impedir que elas sejam definidas ou chamadas. A exclusão de funções de membro especiais fornece uma maneira mais limpa de impedir que o compilador gere funções de membro especiais que você não deseja. A função deve ser excluída ao ser declarada; não pode ser excluída posteriormente, da maneira como uma função pode ser declarada e depois usada como padrão mais tarde.

```
struct widget
{
    // deleted operator new prevents widget from being dynamically allocated.
    void* operator new(std::size_t) = delete;
};
```

A exclusão de uma função de membro normal ou de funções de não membro impede que promoções de tipos problemáticos façam com que uma função não intencional seja chamada. Isso funciona porque as funções excluídas ainda participam da resolução de sobrecarga e fornecem uma correspondência melhor do que a função que poderia ser chamada após a promoção dos tipos. A chamada de função é resolvida para a função mais específica, mas excluída, e causa um erro do compilador.

```
// deleted overload prevents call through type promotion of float to double from succeeding.
void call_with_true_double_only(float) =delete;
void call_with_true_double_only(double param) { return; }
```

Observe, no exemplo anterior, que chamar `call_with_true_double_only` usando um `float` argumento causaria um erro de compilador, mas chamar `call_with_true_double_only` usando um `int` argumento não seria; no `int` caso, o argumento será promovido de `int` para `double` e chamará com êxito a `double` versão da função, embora isso possa não ser o que é pretendido. Para garantir que qualquer chamada para essa função usando um argumento não duplo cause um erro do compilador, você pode declarar uma versão de modelo da função que foi excluída.

```
template < typename T >
void call_with_true_double_only(T) =delete; //prevent call through type promotion of any T to double from
// succeeding.

void call_with_true_double_only(double param) { return; } // also define for const double, double&, etc. as
// needed.
```

Pesquisa de nome dependente do argumento (Koenig) em funções

25/03/2020 • 2 minutes to read • [Edit Online](#)

O compilador pode usar a pesquisa por nome dependente de argumento para localizar a definição de uma chamada de função não qualificada. A pesquisa por nome dependente de argumento também é chamada de pesquisa de Koenig. O tipo de cada argumento em uma chamada de função é definido dentro de uma hierarquia de namespaces, classes, estruturas, uniões ou modelos. Quando você especifica uma chamada de função de sufixo não qualificado, o compilador pesquisa a definição de função na hierarquia associada a cada tipo de argumento.

Exemplo

No exemplo, o compilador nota que a função `f()` pega um argumento `x`. O argumento `x` é do tipo `A::x`, que é definido no namespace `A`. O compilador pesquisa o namespace `A` e encontra uma definição para a função `f()`, que usa um argumento do tipo `A::x`.

```
// argument_dependent_name_koenig_lookup_on_functions.cpp
namespace A
{
    struct X
    {
    };
    void f(const X&)
    {
    }
}
int main()
{
    // The compiler finds A::f() in namespace A, which is where
    // the type of argument x is defined. The type of x is A::X.
    A::X x;
    f(x);
}
```

Argumentos padrão

02/09/2020 • 3 minutes to read • [Edit Online](#)

Em muitos casos, as funções têm os argumentos que são usados com tão raramente que um valor padrão bastaria. Para resolver isso, a facilidade do argumento padrão permite especificar apenas os argumentos de uma função que são significativos em uma determinada chamada. Para ilustrar esse conceito, considere o exemplo apresentado em [sobrecarga de função](#).

```
// Prototype three print functions.
int print( char *s );                                // Print a string.
int print( double dvalue );                          // Print a double.
int print( double dvalue, int prec ); // Print a double with a
// given precision.
```

Em muitos aplicativos, uma opção razoável pode ser fornecida para `prec`, eliminando a necessidade de duas funções:

```
// Prototype two print functions.
int print( char *s );                                // Print a string.
int print( double dvalue, int prec=2 ); // Print a double with a
// given precision.
```

A implementação da `print` função é ligeiramente alterada para refletir o fato de que apenas uma dessas funções existe para o tipo `double`:

```
// default_arguments.cpp
// compile with: /EHsc /c

// Print a double in specified precision.
// Positive numbers for precision indicate how many digits
// precision after the decimal point to show. Negative
// numbers for precision indicate where to round the number
// to the left of the decimal point.

#include <iostream>
#include <math.h>
using namespace std;

int print( double dvalue, int prec ) {
    // Use table-lookup for rounding/truncation.
    static const double rgPow10[] = {
        10E-7, 10E-6, 10E-5, 10E-4, 10E-3, 10E-2, 10E-1, 10E0,
        10E1, 10E2, 10E3, 10E4, 10E5, 10E6
    };
    const int iPowZero = 6;
    // If precision out of range, just print the number.
    if( prec >= -6 && prec <= 7 )
        // Scale, truncate, then rescale.
        dvalue = floor( dvalue / rgPow10[iPowZero - prec] ) *
            rgPow10[iPowZero - prec];
    cout << dvalue << endl;
    return cout.good();
}
```

Para invocar a nova função `print`, use o código a seguir:

```
print( d );      // Precision of 2 supplied by default argument.  
print( d, 0 ); // Override default argument to achieve other  
// results.
```

Observe esses pontos ao usar os argumentos padrão:

- Os argumentos padrão são usados somente em chamadas de função onde os argumentos à direita são omitidos — devem ser os últimos argumentos. Dessa forma, o código a seguir é ilegal:

```
int print( double dvalue = 0.0, int prec );
```

- Um argumento padrão não pode ser redefinido nas declarações posteriores mesmo se a redefinição for idêntica ao original. Portanto, o código a seguir gera um erro:

```
// Prototype for print function.  
int print( double dvalue, int prec = 2 );  
  
...  
  
// Definition for print function.  
int print( double dvalue, int prec = 2 )  
{  
    ...  
}
```

O problema com esse código é que a declaração da função na definição redefine o argumento padrão para `prec`.

- Os argumentos padrão adicionais podem ser adicionados por declarações posteriores.
- Os argumentos padrão podem ser fornecidos para ponteiros para as funções. Por exemplo:

```
int (*pShowIntVal)( int i = 0 );
```

Funções embutidas (C++)

02/09/2020 • 14 minutes to read • [Edit Online](#)

Uma função definida no corpo de uma declaração de classe é uma função embutida.

Exemplo

Na seguinte declaração de classe, o construtor `Account` é uma função embutida. As funções member `GetBalance`, `Deposit`, e `Withdraw` não são especificadas como `inline`, mas podem ser implementadas como funções embutidas.

```
// Inline_Member_Functions.cpp
class Account
{
public:
    Account(double initial_balance) { balance = initial_balance; }
    double GetBalance();
    double Deposit( double Amount );
    double Withdraw( double Amount );
private:
    double balance;
};

inline double Account::GetBalance()
{
    return balance;
}

inline double Account::Deposit( double Amount )
{
    return ( balance += Amount );
}

inline double Account::Withdraw( double Amount )
{
    return ( balance -= Amount );
}

int main()
{
```

NOTE

Na declaração de classe, as funções foram declaradas sem a `inline` palavra-chave. A `inline` palavra-chave pode ser especificada na declaração de classe; o resultado é o mesmo.

Uma determinada função de membro embutida deve ser declarada da mesma maneira em cada unidade de compilação. Essa restrição faz com que as funções embutidas se comportem como se fossem funções instanciadas. Além disso, deve haver exatamente uma definição de uma função embutida.

Uma função de membro de classe usa como padrão a ligação externa, a menos que uma definição para essa função contenha o `inline` especificador. O exemplo anterior mostra que você não precisa declarar essas funções explicitamente com o `inline` especificador. Usar `inline` na definição da função faz com que ela seja uma função embutida. No entanto, não é permitido redeclarar uma função como `inline` após uma chamada para essa função.

inline, __inline, e __forceinline

Os `inline` `__inline` especificadores e instruem o compilador a inserir uma cópia do corpo da função em cada lugar em que a função é chamada.

A inserção, chamada de *expansão embutida* ou de *inalinhamento*, ocorre somente se a análise de custo-benefício do compilador mostrar que vale a pena. A expansão embutida minimiza a sobrecarga de chamada de função com o custo potencial do tamanho de código maior.

A `__forceinline` palavra-chave substitui a análise de custo-benefício e conta com o julgamento do programador em vez disso. Tenha cuidado ao usar o `__forceinline`. O uso de indiscriminado de `__forceinline` pode resultar em um código maior com ganhos de desempenho marginal ou, em alguns casos, até perdas de desempenho (devido à maior paginação de um executável maior, por exemplo).

O uso das funções embutidas pode fazer com que seu programa seja mais rápido porque ele elimina a sobrecarga associada às chamadas de função. As funções com expansão embutida estão sujeitas a otimizações de código não disponíveis às funções normais.

O compilador trata as opções de expansão embutida e as palavras-chave como sugestões. Não há garantia de que as funções serão embutidas. Você não pode forçar o compilador a embutir uma função específica, mesmo com a `__forceinline` palavra-chave. Ao compilar com `/clr`, o compilador não embutirá uma função se houver atributos de segurança aplicados à função.

A `inline` palavra-chave está disponível apenas em C++. As `__inline` `__forceinline` palavras-chave e estão disponíveis em C e C++. Para compatibilidade com versões anteriores `__inline` e `__forceinline` são sinônimos para `__inline`, e `__forceinline` a menos que a opção do compilador [/za](#) (desabilite extensões de linguagem) seja especificada.

A `inline` palavra-chave informa ao compilador que a expansão embutida é preferida. No entanto, o compilador pode criar uma instância separada da função (uma instância) e criar vínculos de chamada padrão em vez de inserir o código embutido. Dois casos em que esse comportamento pode ocorrer são:

- Funções recursivas.
- Funções às quais são feita referência por meio de um ponteiro em outro lugar na unidade de tradução.

Esses motivos podem interferir na inlinagem, *como outras podem*, a critério do compilador; Você não deve depender do `inline` especificador para fazer com que uma função seja embutida.

Assim como acontece com funções normais, não há nenhuma ordem definida para avaliação de argumento em uma função embutida. Na verdade, ele pode ser diferente da ordem de avaliação do argumento quando passado usando o protocolo de chamada de função normal.

A `/ob` opção de otimização do compilador ajuda a determinar se a expansão da função embutida realmente ocorre.

`/LTCG` o embutido em módulo faz a inlinagem se ele é solicitado no código-fonte ou não.

Exemplo 1

```
// inline_keyword1.cpp
// compile with: /c
inline int max( int a , int b ) {
    if( a > b )
        return a;
    return b;
}
```

As funções de membro de uma classe podem ser declaradas em linha, seja usando a `inline` palavra-chave ou colocando a definição de função dentro da definição de classe.

Exemplo 2

```
// inline_keyword2.cpp
// compile with: /EHsc /c
#include <iostream>
using namespace std;

class MyClass {
public:
    void print() { cout << i << ' '; } // Implicitly inline
private:
    int i;
};
```

Específico da Microsoft

A `__inline` palavra-chave é equivalente a `inline`.

Mesmo com `__forceinline` o, o compilador não pode embutir código em todas as circunstâncias. O compilador não poderá embutir uma função se:

- A função ou seu chamador é compilado com `/Ob0` (a opção padrão para compilações de depuração).
- A função e o chamador usam tipos diferentes de manipulação de exceções (manipulação de exceções do C++ em uma, manipulação de exceções estruturada no outro).
- A função tem uma lista de argumentos variável.
- A função usa assembly embutido, a menos que seja compilado com `/Ox`, `/O1` ou `/O2`.
- A função é recursiva e não tem um `#pragma inline_recursion(on)` conjunto. Com o pragma, as funções recursivas são embutidas em uma profundidade padrão de 16 chamadas. Para reduzir a profundidade de inalinhanamento, use `inline_depth` pragma.
- A função é virtual e é chamada virtualmente. Chamadas diretas à funções virtuais podem ser embutidas.
- O programa usa o endereço da função e a chamada à função é feita pelo ponteiro. Chamadas diretas a funções que tiveram o endereço removido podem ser embutidas.
- A função também é marcada com o `naked` `__declspec` modificador.

Se o compilador não puder embutir uma função declarada com `__forceinline`, ele gerará um aviso de nível 1, exceto quando:

- A função é compilada usando/OD ou/Ob0. Nenhum inalinhanamento é esperado nesses casos.
- A função é definida externamente, em uma biblioteca incluída ou em outra unidade de tradução, ou é um destino de chamada virtual ou alvo de chamada indireta. O compilador não pode identificar o código não embutido que ele não pode localizar na unidade de tradução atual.

Funções recursivas podem ser substituídas por código embutido em uma profundidade especificada pelo `inline_depth` pragma, até um máximo de 16 chamadas. Após essa profundidade, as chamadas de função recursivas são tratadas como chamadas a uma instância da função. A profundidade até a qual as funções recursivas são examinadas pela heurística embutida não pode exceder 16. O `inline_recursion` pragma controla a expansão embutida de uma função em expansão no momento. Consulte a opção de compilador/Ob ([expansão de função embutida](#)) para obter informações relacionadas.

FINAL específico da Microsoft

Para obter mais informações sobre como usar o `inline` especificador, consulte:

- [Funções membro de classe embutidas](#)
- [Definindo funções C++ embutidas com `dllexport` e `DllImport`](#)

Quando usar funções embutidas

As funções embutidas são usadas da melhor forma para funções pequenas, como acessar membros de dados particulares. A principal finalidade dessas funções de "acessador" de uma ou duas linhas é retornar informações de estado sobre objetos. Funções curtas são sensíveis à sobrecarga de chamadas de função. As funções mais longas gastam proporcionalmente menos tempo na chamada e no retorno da sequência e se beneficiam menos com o uso de inalinhamento.

Uma `Point` classe pode ser definida da seguinte maneira:

```
// when_to_use_inline_functions.cpp
class Point
{
public:
    // Define "accessor" functions as
    // reference types.
    unsigned& x();
    unsigned& y();
private:
    unsigned _x;
    unsigned _y;
};

inline unsigned& Point::x()
{
    return _x;
}
inline unsigned& Point::y()
{
    return _y;
}
int main()
{
```

Pressupondo que a manipulação de coordenadas seja uma operação relativamente comum em um cliente de tal classe, especificar as duas funções de acessador (`x` e `y` no exemplo anterior), como `inline` normalmente economiza a sobrecarga em:

- Chamadas de função (inclusive passagem de parâmetros e colocação do endereço do objeto na pilha)
- Preservação do registro de ativação do chamador
- Configuração do novo quadro de pilhas
- Comunicação do valor de retorno
- Restaurando o quadro de pilha antigo
- Retorno

Funções embutidas versus macros

As funções embutidas são semelhantes às macros, pois o código de função é expandido no ponto da chamada em tempo de compilação. No entanto, as funções embutidas são analisadas pelo compilador e as macros são

expandidas pelo pré-processador. Como resultado, há várias diferenças importantes:

- As funções integradas seguem todos os protocolos de segurança de tipo impostos em funções normais.
- As funções embutidas são especificadas usando a mesma sintaxe de qualquer outra função, exceto que elas incluem a `inline` palavra-chave na declaração da função.
- As expressões transmitidas como argumentos para as funções integradas são avaliadas uma única vez. Em alguns casos, as expressões transmitidas como argumentos para macros podem ser avaliadas mais de uma vez.

O exemplo a seguir mostra uma macro que converte letras minúsculas em maiúsculas:

```
// inline_functions_macro.c
#include <stdio.h>
#include <conio.h>

#define toupper(a) ((a) >= 'a' && ((a) <= 'z') ? ((a)-('a'-'A')):(a)

int main() {
    char ch;
    printf_s("Enter a character: ");
    ch = toupper( getc(stdin) );
    printf_s( "%c", ch );
}

// Sample Input: xyz
// Sample Output: Z
```

A intenção da expressão `toupper(getc(stdin))` é que um caractere deve ser lido do dispositivo de console (`stdin`) e, se necessário, convertido em letras maiúsculas.

Devido à implementação da macro, `getc` é executado uma vez para determinar se o caractere é maior ou igual a "a", e uma vez para determinar se ele é menor ou igual a "z". Se estiver nesse intervalo, `getc` será executado novamente para converter o caractere para letras maiúsculas. Isso significa que o programa aguarda dois ou três caracteres quando, idealmente, deve aguardar apenas um.

As funções integradas corrigem o problema descrito anteriormente:

```
// inline_functions_inline.cpp
#include <stdio.h>
#include <conio.h>

inline char toupper( char a ) {
    return ((a >= 'a' && a <= 'z') ? a-('a'-'A') : a);
}

int main() {
    printf_s("Enter a character: ");
    char ch = toupper( getc(stdin) );
    printf_s( "%c", ch );
}
```

```
Sample Input: a
Sample Output: A
```

Confira também

[noinline](#)

auto_inline

Sobrecarga de operador

02/09/2020 • 5 minutes to read • [Edit Online](#)

A `operator` palavra-chave declara uma função especificando qual *operador-símbolo* significa quando aplicado a instâncias de uma classe. Isso confere ao operador mais de um significado, ou seja, o "sobrecarregado". O compilador distingue entre os diversos significados de um operador examinando os tipos de seus operandos.

Sintaxe

`tipo operator de operador-símbolo(lista de parâmetros)`

Comentários

Você pode redefinir a função da maioria dos operadores internos globalmente ou em uma classe de cada vez. Os operadores sobrecarregados são implementados como funções.

O nome de um operador sobrecarregado é `operator x`, em que `x` é o operador, como aparece na tabela a seguir. Por exemplo, para sobrecarregar o operador de adição, você define uma função chamada `Operator +`. Da mesma forma, para sobrecarregar o operador de adição/atribuição, `+ =` defina uma função chamada `Operator + =`.

Operadores redefiníveis

OPERADOR	NOME	TYPE
,	Vírgula	Binário
!	NOT lógico	Unário
!=	Desigualdade	Binário
%	Modulus	Binário
%=	Atribuição de módulo	Binário
&	AND bit a bit	Binário
&	Address-of	Unário
&&	AND lógico	Binário
&=	Atribuição AND de bit a bit	Binário
()	Chamada de função	—
()	Operador cast	Unário
*	Multiplicação	Binário

OPERADOR	NOME	TYPE
*	Desreferência de ponteiro	Unário
*=	Atribuição de multiplicação	Binário
+	Adição	Binário
+	Mais unário	Unário
++	Incremento ¹	Unário
+=	Atribuição de adição	Binário
-	Subtração	Binário
-	Negação unária	Unário
--	Decrementar ¹	Unário
-=	Atribuição de subtração	Binário
->	Seleção de membro	Binário
->*	Seleção de ponteiro para membro	Binário
/	Divisão	Binário
/=	Atribuição de divisão	Binário
<	Menor que	Binário
<<	Shift esquerda	Binário
<<=	Atribuição de deslocamento para a esquerda	Binário
<=	Menor que ou igual a	Binário
=	Atribuição	Binário
==	Igualitário	Binário
>	Maior que	Binário
>=	Maior ou igual a	Binário
>>	Shift direita	Binário
>>=	Atribuição de deslocamento para a direita	Binário

OPERADOR	NOME	TYPE
[]	Subscrito de matriz	—
^	OR exclusivo	Binário
^=	Atribuição de OR exclusivo	Binário
	OR inclusivo bit a bit	Binário
=	Atribuição OR inclusivo de bit a bit	Binário
	OR lógico	Binário
~	Complemento de um	Unário
<code>delete</code>	Excluir	—
<code>new</code>	Novo	—
operadores de conversão	operadores de conversão	Unário

¹ existem duas versões do incremento unário e os operadores de decréscimo: preincremento e reincrement.

Consulte [regras gerais para sobrecarga de operador](#) para obter mais informações. As restrições nas diversas categorias de operadores sobrecarregados são descritas nos tópicos a seguir:

- [Operadores unários](#)
- [Operadores binários](#)
- [Atribuição](#)
- [Chamada de função](#)
- [Subscrito](#)
- [Acesso de membros de classe](#)
- [Incrementar e decrementar.](#)
- [Conversões de tipo definidas pelo usuário](#)

Os operadores mostrados na tabela a seguir não podem ser sobrecarregados. A tabela inclui os símbolos de pré-processador # e ##.

Operadores não redefiníveis

OPERADOR	NOME
.	Seleção de membro
. *	Seleção de ponteiro para membro
::	Resolução do escopo
? :	Condicional

OPERADOR	NOME
#	Conversão de pré-processador em cadeia de caracteres
##	Concatenação de pré-processador

Embora, de modo geral, os operadores sobrecarregados sejam chamados implicitamente pelo compilador quando são encontrados no código, eles podem ser invocados explicitamente da mesma maneira que qualquer função de membro ou de não membro é chamada:

```
Point pt;
pt.operator+( 3 ); // Call addition operator to add 3 to pt.
```

Exemplo

O exemplo a seguir sobrecarrega o + operador para adicionar dois números complexos e retorna o resultado.

```
// operator_overloading.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

struct Complex {
    Complex( double r, double i ) : re(r), im(i) {}
    Complex operator+( Complex &other );
    void Display( ) { cout << re << ", " << im << endl; }
private:
    double re, im;
};

// Operator overloaded using a member function
Complex Complex::operator+( Complex &other ) {
    return Complex( re + other.re, im + other.im );
}

int main() {
    Complex a = Complex( 1.2, 3.4 );
    Complex b = Complex( 5.6, 7.8 );
    Complex c = Complex( 0.0, 0.0 );

    c = a + b;
    c.Display();
}
```

6.8, 11.2

Nesta seção

- [Regras gerais para sobrecarga de operador](#)
- [Sobrecarregando operadores unários](#)
- [Operadores binários](#)
- [Atribuição](#)
- [Chamada de função](#)

- Subscrito
- Acesso de membro

Confira também

[Operadores internos C++, precedência e associatividade](#)

[Palavras-chave](#)

Regras gerais para sobrecarga de operador

02/09/2020 • 5 minutes to read • [Edit Online](#)

As seguintes regras restringem o modo como os operadores sobrecarregados são implementados. No entanto, eles não se aplicam aos operadores **novo** e **excluir**, que são cobertos separadamente.

- Você não pode definir novos operadores, como ..
- Você não pode redefinir o significado dos operadores quando aplicados aos tipos de dados internos.
- Os operadores sobrecarregados devem ser uma função membro da classe não estática ou uma função global. Uma função global que exige acesso a membros de classe particulares ou protegidos deve ser declarada como um amigo daquela classe. Uma função global deve ter pelo menos um argumento que é do tipo da classe ou do tipo enumerado, ou que é uma referência a uma classe ou a um tipo enumerado. Por exemplo:

```
// rules_for_operator_overloading.cpp
class Point
{
public:
    Point operator<( Point & );    // Declare a member operator
                                    // overload.
    // Declare addition operators.
    friend Point operator+( Point&, int );
    friend Point operator+( int, Point& );
};

int main()
{
```

O exemplo de código anterior declara o operador menor que como uma função membro; no entanto, os operadores de adição são declarados como funções globais que têm acesso de amigo. Observe que mais de uma implementação pode ser fornecida para um determinado operador. No caso do operador de adição acima, as duas implementações são fornecidas para facilitar a comutatividade. É muito provável que os operadores que adicionam um `Point` a um `Point`, `int` um `Point`, e assim por diante, possam ser implementados.

- Os operadores obedecem a precedência, agrupamento e número de operandos ditados por seu uso típico com tipos internos. Portanto, não há como expressar o conceito "adicionar 2 e 3 a um objeto do tipo `Point`", esperando 2 para serem adicionados à coordenada *x* e 3 para serem adicionados à coordenada *y*.
- Os operadores unários declarados como funções membro não pegam argumentos; se declarados como funções globais, eles pegam um argumento.
- Os operadores binários declarados como funções membro pegam um argumento; se declarados como funções globais, eles pegam dois argumentos.
- Se um operador puder ser usado como um operador unário ou binário (`&` , `*` , `+` e `-`), você poderá sobrecarregar cada uso separadamente.
- Os operadores sobrecarregados não podem ter argumentos padrão.
- Todos os operadores sobrecarregados, exceto atribuição (`operador =`), são herdados por classes derivadas.

- O primeiro argumento para operadores sobrecarregados da função membro sempre é do tipo de classe do objeto para o qual o operador é invocado (a classe na qual o operador é declarado, ou uma classe derivada dessa classe.) Nenhuma conversão é fornecida para o primeiro argumento.

Observe que o significado de qualquer operador pode ser alterado completamente. Isso inclui o significado do endereço (`&`), atribuição (`=`) e operadores de chamada de função. Além disso, as identidades que podem ser confiáveis para os tipos internos podem ser modificadas usando a sobrecarga do operador. Por exemplo, as quatro instruções a seguir geralmente são equivalentes quando avaliada completamente:

```
var = var + 1;  
var += 1;  
var++;  
++var;
```

Essa identidade não pode ser confiável para os tipos da classe que sobrecarregam os operadores. Além disso, alguns dos requisitos implícitos no uso desses operadores para tipos básicos são relaxados para operadores sobrecarregados. Por exemplo, o operador de adição/atribuição, `+=` , requer que o operando esquerdo seja um L-Value quando aplicado a tipos básicos; não há tal exigência quando o operador está sobrecarregado.

NOTE

Para consistência, geralmente é melhor seguir o modelo dos tipos internos ao definir operadores sobrecarregados. Se a semântica de um operador sobrecarregado for significativamente diferentes do seu significado em outros contextos, ela pode ser mais confusa do que útil.

Confira também

[Sobrecarga de operador](#)

Operadores unários de sobrecarga

02/09/2020 • 2 minutes to read • [Edit Online](#)

Os operadores unários que podem ser sobrecarregados são os seguintes:

1. `!` (não lógico)
2. `&` (endereço)
3. `~` (complemento de um)
4. `*` (desreferência do ponteiro)
5. `+` (mais unário)
6. `-` (negação unário)
7. `++` (incremento)
8. `--` (decrementar)
9. operadores de conversão

Os operadores de incremento e decréscimo de sufixo (`++` e `--`) são tratados separadamente em [incrementos e decremento](#).

Os operadores de conversão também são discutidos em um tópico separado; Confira [conversões de tipo definidas pelo usuário](#).

As regras a seguir são verdadeiras para todos os outros operadores unários. Para declarar uma função de operador unário como um membro não estático, você deve declará-la na forma:

```
RET-tipo operator op()
```

em que *RET-Type* é o tipo de retorno e *op* é um dos operadores listados na tabela anterior.

Para declarar uma função de operador unário como uma função global, você deve declará-la na forma:

```
RET-tipo operator op( ARG)
```

em que *RET-Type* e *op* são os descritos para as funções de operador de membro e o *ARG* é um argumento do tipo de classe no qual operar.

NOTE

Não há nenhuma restrição quanto aos tipos de retorno dos operadores unários. Por exemplo, faz sentido para NOT lógico (`!`) retornar um valor integral, mas isso não é imposto.

Confira também

[Sobrecarga de operador](#)

Sobrecarga dos operadores de incremento e decremento (C++)

02/09/2020 • 3 minutes to read • [Edit Online](#)

Os operadores de incremento e decremento formam uma categoria especial porque há duas variantes de cada um:

- Pré-incremento e pós-incremento
- Pré-decremento e pós-decremento

Ao escrever funções de operador sobrecarregado, pode ser útil implementar versões separadas para as versões pré-fixada e pós-fixada desses operadores. Para distinguir entre os dois, a seguinte regra é observada: a forma de prefixo do operador é declarada exatamente da mesma forma que qualquer outro operador unário; o formulário sufixo aceita um argumento adicional do tipo `int`.

NOTE

Ao especificar um operador sobrecarregado para a forma de sufixo do operador de incremento ou decréscimo, o argumento adicional deve ser do tipo `int`; especificar qualquer outro tipo gera um erro.

O exemplo a seguir mostra como definir operadores de incremento e decremento pré-fixados e pós-fixados para a classe `Point`:

```

// increment_and_decrement1.cpp
class Point
{
public:
    // Declare prefix and postfix increment operators.
    Point& operator++();           // Prefix increment operator.
    Point operator++(int);         // Postfix increment operator.

    // Declare prefix and postfix decrement operators.
    Point& operator--();           // Prefix decrement operator.
    Point operator--(int);         // Postfix decrement operator.

    // Define default constructor.
    Point() { _x = _y = 0; }

    // Define accessor functions.
    int x() { return _x; }
    int y() { return _y; }
private:
    int _x, _y;
};

// Define prefix increment operator.
Point& Point::operator++()
{
    _x++;
    _y++;
    return *this;
}

// Define postfix increment operator.
Point Point::operator++(int)
{
    Point temp = *this;
    ++*this;
    return temp;
}

// Define prefix decrement operator.
Point& Point::operator--()
{
    _x--;
    _y--;
    return *this;
}

// Define postfix decrement operator.
Point Point::operator--(int)
{
    Point temp = *this;
    --*this;
    return temp;
}

int main()
{
}

```

Os mesmos operadores podem ser definidos no escopo do arquivo (globalmente) usando os seguintes cabeçalhos de função:

```

friend Point& operator++( Point& )           // Prefix increment
friend Point& operator++( Point&, int ) // Postfix increment
friend Point& operator--( Point& )           // Prefix decrement
friend Point& operator--( Point&, int ) // Postfix decrement

```

O argumento do tipo `int` que denota a forma de sufixo do operador de incremento ou decréscimo não é comumente usado para passar argumentos. Em geral, ele contém o valor 0. No entanto, pode ser usado como segue:

```
// increment_and_decrement2.cpp
class Int
{
public:
    Int &operator++( int n );
private:
    int _i;
};

Int& Int::operator++( int n )
{
    if( n != 0 )    // Handle case where an argument is passed.
        _i += n;
    else
        _i++;        // Handle case where no argument is passed.
    return *this;
}
int main()
{
    Int i;
    i.operator++( 25 ); // Increment by 25.
}
```

Não há nenhuma outra sintaxe para usar os operadores de incremento ou decremento para passar esses valores que não seja a invocação explícita, conforme mostrado no código acima. Uma maneira mais simples de implementar essa funcionalidade é sobrepor o operador de adição/atribuição (`+=`).

Confira também

[Sobrepor de operador](#)

Operadores binários

02/09/2020 • 2 minutes to read • [Edit Online](#)

A tabela a seguir mostra uma lista de operadores que podem ser sobre carregados.

Operadores binários redefiníveis

OPERADOR	NOME
,	Vírgula
!=	Desigualdade
%	Modulus
%=	Módulo/atribuição
&	AND bit a bit
&&	AND lógico
&=	Atribuição AND bit a bit
*	Multiplicação
*=	Atribuição/multiplicação
+	Adição
+=	Atribuição/adição
-	Subtração
-=	Subtração/atribuição
->	Seleção de membro
->*	Seleção de ponteiro para membro
/	Divisão
/=	Divisão/atribuição
<	Menor que
<<	Shift esquerda
<<=	Deslocamento para a esquerda/atribuição

OPERADOR	NOME
<code><=</code>	Menor que ou igual a
<code>=</code>	Atribuição
<code>==</code>	Igualitário
<code>></code>	Maior que
<code>>=</code>	Maior que ou igual a
<code>>></code>	Shift direita
<code>>>=</code>	Deslocamento para direita/atribuição
<code>^</code>	OR exclusivo
<code>^=</code>	Atribuição OR exclusivo
<code> </code>	OR inclusivo bit a bit
<code> =</code>	OR inclusivo bit a bit/atribuição
<code> </code>	OR lógico

Para declarar uma função de operador binário como um membro não estático, você deve declará-la na forma:

`RET-tipo operator op(ARG)`

em que *RET-Type* é o tipo de retorno, *op* é um dos operadores listados na tabela anterior e *ARG* é um argumento de qualquer tipo.

Para declarar uma função de operador binário como uma função global, você deve declará-la na forma:

`RET-tipo operator op(arg1, arg2)`

em que *RET-Type* e *op* são descritos como descrito para funções de operador de membro e *arg1* e *arg2* são argumentos. Ao menos um dos argumentos deve ser do tipo da classe.

NOTE

Não há nenhuma restrição quanto aos tipos de retorno dos operadores binários; no entanto, a maioria dos operadores binários definidos pelo usuário retornam um tipo de classe ou uma referência a um tipo de classe.

Confira também

[Sobrecarga de operador](#)

Atribuição

25/03/2020 • 2 minutes to read • [Edit Online](#)

O operador de atribuição (=) é, estritamente falando, um operador binário. Sua declaração é idêntica a qualquer outro operador binário, com as seguintes exceções:

- Ele deve ser uma função de membro não estático. Nenhum **operador** = pode ser declarado como uma função não membro.
- Ele não é herdado por classes derivadas.
- Uma função **operador** = padrão pode ser gerada pelo compilador para tipos de classe, se não houver nenhum.

O exemplo a seguir ilustra como declarar um operador de atribuição:

```
class Point
{
public:
    int _x, _y;

    // Right side of copy assignment is the argument.
    Point& operator=(const Point&);

};

// Define copy assignment operator.
Point& Point::operator=(const Point& otherPoint)
{
    _x = otherPoint._x;
    _y = otherPoint._y;

    // Assignment operator returns left side of assignment.
    return *this;
}

int main()
{
    Point pt1, pt2;
    pt1 = pt2;
}
```

O argumento fornecido é o lado direito da expressão. O operador retorna o objeto para preservar o comportamento do operador de atribuição, que retorna o valor do lado esquerdo após a conclusão da atribuição. Isso permite o encadeamento de atribuições, como:

```
pt1 = pt2 = pt3;
```

O operador de atribuição de cópia não deve ser confundido com o construtor de cópia. O último é chamado durante a construção de um novo objeto a partir de um existente:

```
// Copy constructor is called--not overloaded copy assignment operator!
Point pt3 = pt1;

// The previous initialization is similar to the following:
Point pt4(pt1); // Copy constructor call.
```

NOTE

É aconselhável seguir a [regra de três](#) que uma classe que define um operador de atribuição de cópia também deve definir explicitamente o construtor de cópia, o destruidor e, começando com o C++ 11, mover o construtor e mover o operador de atribuição.

Confira também

- [Sobrecarga de Operador](#)
- [Operadores de construtores de cópia e de atribuição de cópia \(C++\)](#)

Chamada de função (C++)

25/03/2020 • 2 minutes to read • [Edit Online](#)

O operador de chamada de função, invocado usando parênteses, é um operador binário.

Sintaxe

```
primary-expression ( expression-list )
```

Comentários

Neste contexto, `primary-expression` é o primeiro operando e `expression-list`, uma possível lista de argumentos vazia, é o segundo. O operador de chamada de função é usado para operações que exigem um número de parâmetros. Isso funciona porque `expression-list` é uma lista em vez de um operando único. O operador de chamada de função deve ser uma função de membro não estático.

O operador de chamada de função, quando sobrecarregado, não modifica o modo como as funções são chamadas; modifica o modo como o operador deve ser interpretado quando aplicado aos objetos de um tipo de classe. Por exemplo, o código a seguir normalmente não teria sentido:

```
Point pt;
pt( 3, 2 );
```

Entretanto, com um operador de chamada de função sobrecarregado apropriado, essa sintaxe pode ser usada para deslocar a coordenada `x` 3 unidades e a coordenada `y` 2 unidades. O código a seguir mostra essa definição:

```
// function_call.cpp
class Point
{
public:
    Point() { _x = _y = 0; }
    Point &operator()( int dx, int dy )
    { _x += dx; _y += dy; return *this; }
private:
    int _x, _y;
};

int main()
{
    Point pt;
    pt( 3, 2 );
}
```

Observe que o operador de chamada de função é aplicado ao nome de um objeto, não ao nome de uma função.

Você pode também sobrecarregar o operador de chamada de função usando um ponteiro para a função (em vez da própria função).

```
typedef void(*ptf)();
void func()
{
}
struct S
{
    operator ptf()
    {
        return func;
    }
};

int main()
{
    S s;
    s(); //operates as s.operator ptf()()
}
```

Confira também

[Sobrecarga de Operador](#)

Subscrito

02/09/2020 • 2 minutes to read • [Edit Online](#)

O operador de subscrito ([]), como o operador de chamada de função, é considerado um operador binário. O operador subscrito deve ser uma função de membro não estático que usa um único argumento. Este argumento ou pode ser de qualquer tipo e designa o subscrito de matriz desejado.

Exemplo

O exemplo a seguir demonstra como criar um vetor do tipo `int` que implementa a verificação de limites:

```
// subscripting.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
class IntVector {
public:
    IntVector( int cElements );
    ~IntVector() { delete [] _iElements; }
    int& operator[](int nSubscript);
private:
    int *_iElements;
    int _iUpperBound;
};

// Construct an IntVector.
IntVector::IntVector( int cElements ) {
    _iElements = new int[cElements];
    _iUpperBound = cElements;
}

// Subscript operator for IntVector.
int& IntVector::operator[](int nSubscript) {
    static int iErr = -1;

    if( nSubscript >= 0 && nSubscript < _iUpperBound )
        return _iElements[nSubscript];
    else {
        clog << "Array bounds violation." << endl;
        return iErr;
    }
}

// Test the IntVector class.
int main() {
    IntVector v( 10 );
    int i;

    for( i = 0; i <= 10; ++i )
        v[i] = i;

    v[3] = v[9];

    for ( i = 0; i <= 10; ++i )
        cout << "Element: [" << i << "] = " << v[i] << endl;
}
```

```
Array bounds violation.  
Element: [0] = 0  
Element: [1] = 1  
Element: [2] = 2  
Element: [3] = 9  
Element: [4] = 4  
Element: [5] = 5  
Element: [6] = 6  
Element: [7] = 7  
Element: [8] = 8  
Element: [9] = 9  
Array bounds violation.  
Element: [10] = 10
```

Comentários

Quando `i` atinge 10 no programa anterior, **Operator []** detecta que um subscrito fora dos limites está sendo usado e emite uma mensagem de erro.

Observe que o **operador function []** retorna um tipo de referência. Isso a torna um l-value, permitindo usar as expressões subscritas em ambos os lados dos operadores de atribuição.

Confira também

[Sobrecarga de operador](#)

Acesso de membro

25/03/2020 • 2 minutes to read • [Edit Online](#)

O acesso de membro de classe pode ser controlado pela sobrecarga do operador de acesso de membro (->). Esse operador é considerado um operador unário nesse uso, e a função sobrecarregada do operador deve ser uma função de membro da classe. Portanto, a declaração dessa função é:

Sintaxe

```
class-type *operator->()
```

Comentários

em que *tipo de classe* é o nome da classe à qual esse operador pertence. A função do operador de acesso do membro deve ser uma função de membro não estático.

Esse operador é usado (frequentemente em conjunto com o operador de desreferência de ponteiro) para implementar "ponteiros inteligentes" que validam os ponteiros antes da desreferência ou do uso da contagem.

O elemento de linguagem . o operador de acesso de membro não pode ser sobrecarregado.

Confira também

[Sobrecarga de Operador](#)

Classes e structs (C++)

02/09/2020 • 2 minutes to read • [Edit Online](#)

Esta seção apresenta classes e estruturas do C++. As duas construções são idênticas em C++, exceto que, em structs, a acessibilidade padrão é pública, enquanto em classes o padrão é privado.

Classes e structs são as construções pelas quais você define seus próprios tipos. Classes e structs podem conter membros de dados e funções de membro, que permitem descrever o estado e o comportamento do tipo.

Os tópicos a seguir estão incluídos:

- [class](#)
- [struct](#)
- [Visão geral do membro da classe](#)
- [Controle de acesso de membro](#)
- [Herança](#)
- [Membros estáticos](#)
- [Conversões de tipo definidas pelo usuário](#)
- [Membros de dados mutáveis \(especificador mutável\)](#)
- [Declarações de classe aninhadas](#)
- [Tipos de classe anônima](#)
- [Ponteiros para membros](#)
- [Esse ponteiro](#)
- [Campos de bits do C++](#)

Os três tipos de classe são estrutura, classe e união. Eles são declarados usando as palavras-chave [struct](#), [Classe](#) [Union](#). A tabela a seguir mostra as diferenças entre os três tipos de classe.

Para obter mais informações sobre uniões, consulte [uniões](#). Para obter informações sobre classes e structs em C++/CLI e C++/CX, consulte [classes e estruturas](#).

Controle de acesso e restrições de estruturas, classes e uniões

ESTRUTURAS	CLASSES	UNIÕES
a chave de classe é** <code>struct</code> **	a chave de classe é** <code>class</code> **	a chave de classe é** <code>union</code> **
O acesso padrão é público	O acesso padrão é particular	O acesso padrão é público
Nenhuma restrição de uso	Nenhuma restrição de uso	Use apenas um membro de cada vez

Confira também

[Referência da linguagem C++](#)

class (C++)

02/09/2020 • 3 minutes to read • [Edit Online](#)

A `class` palavra-chave declara um tipo de classe ou define um objeto de um tipo de classe.

Sintaxe

```
[template-spec]
class [ms-decl-spec] [tag [: base-list ]]
{
    member-list
} [declarators];
[ class ] tag declarators;
```

parâmetros

modelo-especificação

Especificações de modelo opcionais. Para obter mais informações, consulte [modelos](#).

class

A `class` palavra-chave.

MS-decl-spec

Especificação de classe de armazenamento opcional. Para obter mais informações, consulte a palavra-chave [__declspec](#).

marcação

O nome do tipo dado à classe. A marca se torna uma palavra reservada no escopo da classe. A marca é opcional. Se omitida, uma classe anônima será definida. Para obter mais informações, consulte [tipos de classe anônima](#).

base-list

A lista opcional de classes ou estruturas da qual esta classe derivará seus membros. Consulte [classes base](#) para obter mais informações. Cada classe base ou nome de estrutura pode ser precedido por um especificador de acesso ([público](#), [privado](#), [protegido](#)) e a palavra-chave [virtual](#). Consulte a tabela de acesso de membro em [controlando o acesso a membros de classe](#) para obter mais informações.

lista de membros

Lista de membros da classe. Consulte [visão geral de membro de classe](#) para obter mais informações.

declarators

Lista de declaradores que especifica os nomes de uma ou mais instâncias do tipo da classe. Os declaradores podem incluir listas de inicializador se todos os membros de dados da classe forem `public`. Isso é mais comum em estruturas, cujos membros de dados são `public` por padrão, do que em classes. Consulte [visão geral dos declaradores](#) para obter mais informações.

Comentários

Para obter mais informações sobre as classes em geral, consulte um dos seguintes tópicos:

- [struct](#)
- [union](#)
- [__multiple_inheritance](#)

- [single_inheritance](#)
- [virtual_inheritance](#)

Para obter informações sobre classes e estruturas gerenciadas em C++/CLI e C++/CX, consulte [classes e structs](#)

Exemplo

```
// class.cpp
// compile with: /EHsc
// Example of the class keyword
// Exhibits polymorphism/virtual functions.

#include <iostream>
#include <string>
using namespace std;

class dog
{
public:
    dog()
    {
        _legs = 4;
        _bark = true;
    }

    void setDogSize(string dogSize)
    {
        _dogSize = dogSize;
    }
    virtual void setEars(string type)      // virtual function
    {
        _earType = type;
    }

private:
    string _dogSize, _earType;
    int _legs;
    bool _bark;

};

class breed : public dog
{
public:
    breed( string color, string size)
    {
        _color = color;
        setDogSize(size);
    }

    string getColor()
    {
        return _color;
    }

    // virtual function redefined
    void setEars(string length, string type)
    {
        _earLength = length;
        _earType = type;
    }

protected:
    string _color, _earLength, _earType;
};
```

```
int main()
{
    dog mongrel;
    breed labrador("yellow", "large");
    mongrel.setEars("pointy");
    labrador.setEars("long", "floppy");
    cout << "Cody is a " << labrador.getColor() << " labrador" << endl;
}
```

Confira também

[Palavras-chave](#)

[Classes e structs](#)

struct (C++)

02/09/2020 • 4 minutes to read • [Edit Online](#)

A `struct` palavra-chave define um tipo de estrutura e/ou uma variável de um tipo de estrutura.

Sintaxe

```
[template-spec] struct [ms-decl-spec] [tag [: base-list ]]
{
    member-list
} [declarators];
[struct] tag declarators;
```

parâmetros

modelo-especificação

Especificações de modelo opcionais. Para obter mais informações, consulte [especificações de modelo](#).

struct

A `struct` palavra-chave.

MS-decl-spec

Especificação de classe de armazenamento opcional. Para obter mais informações, consulte a palavra-chave [_declspec](#).

marcação

O nome do tipo dado à estrutura. A marca se torna uma palavra reservada no escopo da estrutura. A marca é opcional. Se omitida, uma estrutura anônima será definida. Para obter mais informações, consulte [tipos de classe anônima](#).

base-list

A lista opcional de classes ou estruturas de que esta estrutura derivará seus membros. Consulte [classes base](#) para obter mais informações. Cada classe base ou nome de estrutura pode ser precedido por um especificador de acesso ([público](#), [privado](#), [protegido](#)) e a palavra-chave [virtual](#). Consulte a tabela de acesso de membro em [controlando o acesso a membros de classe](#) para obter mais informações.

lista de membros

Lista de membros da estrutura. Consulte [visão geral de membro de classe](#) para obter mais informações. A única diferença aqui é que `struct` é usada no lugar de `class`.

declarators

Lista de declaradores especificando os nomes da estrutura. As listas de declaradores declaram uma ou mais instâncias do tipo de estrutura. Os declaradores podem incluir listas de inicializador se todos os membros de dados da estrutura forem `public`. As listas de inicializadores são comuns em estruturas porque os membros de dados são `public` por padrão. Consulte [visão geral dos declaradores](#) para obter mais informações.

Comentários

Um tipo de estrutura é um tipo composto definido pelo usuário. É composto de campos ou membros que podem ter tipos diferentes.

Em C++, uma estrutura é igual a uma classe, exceto que seus membros são `public` por padrão.

Para obter informações sobre classes e estruturas gerenciadas em C++/CLI, consulte [classes e estruturas](#).

Usando uma estrutura

Em C, você deve usar explicitamente a `struct` palavra-chave para declarar uma estrutura. Em C++, você não precisa usar a `struct` palavra-chave após a definição do tipo.

Há também a opção de declarar variáveis quando o tipo de estrutura é definido colocando um ou vários nomes de variável separados por vírgulas entre a chave de fechamento e o ponto-e-vírgula.

As variáveis de estrutura podem ser inicializadas. A inicialização de cada variável deve ser incluída entre chaves.

Para obter informações relacionadas, consulte [Class, Union e enum](#).

Exemplo

```
#include <iostream>
using namespace std;

struct PERSON { // Declare PERSON struct type
    int age; // Declare member types
    long ss;
    float weight;
    char name[25];
} family_member; // Define object of type PERSON

struct CELL { // Declare CELL bit field
    unsigned short character : 8; // 00000000 ???????
    unsigned short foreground : 3; // 00000??? 00000000
    unsigned short intensity : 1; // 0000?000 00000000
    unsigned short background : 3; // 0??0000 00000000
    unsigned short blink : 1; // ?000000 00000000
} screen[25][80]; // Array of bit fields

int main() {
    struct PERSON sister; // C style structure declaration
    PERSON brother; // C++ style structure declaration
    sister.age = 13; // assign values to members
    brother.age = 7;
    cout << "sister.age = " << sister.age << '\n';
    cout << "brother.age = " << brother.age << '\n';

    CELL my_cell;
    my_cell.character = 1;
    cout << "my_cell.character = " << my_cell.character;
}
// Output:
// sister.age = 13
// brother.age = 7
// my_cell.character = 1
```

Visão geral de membros de classe

02/09/2020 • 6 minutes to read • [Edit Online](#)

Uma classe ou estrutura consiste em seus membros. O trabalho que uma classe faz é executado por suas funções de membro. O estado que ele mantém é armazenado em seus membros de dados. A inicialização de membros é feita por construtores e o trabalho de limpeza, como a liberação de memória e a liberação de recursos, é feito por destruidores. No C++ 11 e posterior, os membros de dados podem (e geralmente devem) ser inicializados no ponto de declaração.

Tipos de membros de classe

A lista completa de categorias de membros é a seguinte:

- [Funções de membro especiais.](#)
- [Visão geral das funções de membro.](#)
- [Membros de dados](#), incluindo tipos internos e outros tipos definidos pelo usuário.
- Operadores
- [Declarações de classe aninhadas](#) e.)
- [Uniões](#)
- [Enumerações.](#)
- [Campos de bits.](#)
- [Amigos.](#)
- [Aliases e TYPEDEFs.](#)

NOTE

Os friends são incluídos na lista anterior porque estão contidos na declaração da classe. Porém, eles não são membros da classe true, pois não estão no escopo da classe.

Declaração de classe de exemplo

O exemplo a seguir mostra uma declaração de classe simples:

```

// TestRun.h

class TestRun
{
    // Start member list.

    //The class interface accessible to all callers.
public:
    // Use compiler-generated default constructor:
    TestRun() = default;
    // Don't generate a copy constructor:
    TestRun(const TestRun&) = delete;
    TestRun(std::string name);
    void DoSomething();
    int Calculate(int a, double d);
    virtual ~TestRun();
    enum class State { Active, Suspended };

    // Accessible to this class and derived classes only.
protected:
    virtual void Initialize();
    virtual void Suspend();
    State GetState();

    // Accessible to this class only.
private:
    // Default brace-initialization of instance members:
    State _state{ State::Suspended };
    std::string _testName{ "" };
    int _index{ 0 };

    // Non-const static member:
    static int _instances;
    // End member list.
};

// Define and initialize static member.
int TestRun::_instances{ 0 };

```

Acessibilidade de membro

Os membros de uma classe são declarados na lista de membros. A lista de membros de uma classe pode ser dividida em qualquer número `private` de `protected` e `public` seções usando palavras-chave conhecidas como especificadores de acesso. Dois-pontos : deve seguir o especificador de acesso. Essas seções não precisam ser contíguas, ou seja, qualquer uma dessas palavras-chave pode aparecer várias vezes na lista de membros. A palavra-chave designa o acesso de todos os membros acima até o próximo especificador de acesso ou a próxima chave de fechamento. Para obter mais informações, consulte [controle de acesso de membro \(C++\)](#).

Membros estáticos

Um membro de dados pode ser declarado como estático, o que significa que todos os objetos da classe têm acesso à mesma cópia dele. Uma função de membro pode ser declarada como estática; nesse caso, ela só pode acessar membros de dados estáticos da classe (e não tem *esse* ponteiro). Para obter mais informações, consulte [membros de dados estáticos](#).

Funções de membro especiais

Funções de membro especiais são funções que são fornecidas automaticamente pelo compilador se você não especificá-las em seu código-fonte.

1. Construtor padrão
2. Construtor de cópia
3. (C++ 11) Mover Construtor
4. Operador de atribuição de cópia
5. (C++ 11) Mover operador de atribuição
6. Destruídos

Para obter mais informações, consulte [funções de membro especiais](#).

Inicialização do memberwise

No C++ 11 e posterior, os declaradores de membro não estáticos podem conter inicializadores.

```
class CanInit
{
public:
    long num {7};           // OK in C++11
    int k = 9;              // OK in C++11
    static int i = 9; // Error: must be defined and initialized
                      // outside of class declaration.

    // initializes num to 7 and k to 9
    CanInit(){}

    // overwrites original initialized value of num:
    CanInit(int val) : num(val) {}

};

int main()
{}
```

Se um membro for atribuído a um valor em um construtor, esse valor substituirá o valor com o qual o membro foi inicializado no ponto de declaração.

Há apenas uma cópia compartilhada membros de dados estáticos para todos os objetos de um determinado tipo de classe. Os membros de dados estáticos devem ser definidos e podem ser inicializados no escopo do arquivo. (Para obter mais informações sobre membros de dados estáticos, consulte [membros de dados estáticos](#).) O exemplo a seguir mostra como executar essas inicializações:

```
// class_members2.cpp
class CanInit2
{
public:
    CanInit2() {} // Initializes num to 7 when new objects of type
                  // CanInit are created.
    long      num {7};
    static int i;
    static int j;
};

// At file scope:

// i is defined at file scope and initialized to 15.
// The initializer is evaluated in the scope of CanInit.
int CanInit2::i = 15;

// The right side of the initializer is in the scope
// of the object being initialized
int CanInit2::j = i;
```

NOTE

O nome da classe, `CanInit2`, deve preceder `i` para especificar que `i` que está sendo definida é um membro da classe `CanInit2`.

Confira também

[Classes e structs](#)

Controle de acesso a membro (C++)

02/09/2020 • 11 minutes to read • [Edit Online](#)

Os controles de acesso permitem que você separe a interface **pública** de uma classe dos detalhes de implementação **privada** e os membros **protegidos** que são apenas para uso por classes derivadas. O especificador de acesso se aplica a todos os membros declarados depois dele até que o próximo especificador de acesso seja encontrado.

```
class Point
{
public:
    Point( int, int ) // Declare public constructor. ;
    Point(); // Declare public default constructor.
    int &x( int ); // Declare public accessor.
    int &y( int ); // Declare public accessor.

private:           // Declare private state variables.
    int _x;
    int _y;

protected:        // Declare protected function for derived classes only.
    Point ToWindowCoords();
};
```

O acesso padrão está **private** em uma classe e **public** em uma struct ou União. Especificadores de acesso em uma classe podem ser usados várias vezes em qualquer ordem. A alocação de armazenamento para objetos de tipos de classe depende da implementação, mas é garantido que os membros receberão endereços de memória sucessivamente mais altos entre especificadores de acesso.

Controle de acesso de membros

TIPO DE ACESSO	SIGNIFICADO
pessoal	Membros de classe declarados como private podem ser usados somente por funções de membro e amigos (classes ou funções) da classe.
protected	Membros de classe declarados como protected podem ser usados por funções de membro e amigos (classes ou funções) da classe. Além disso, eles podem ser usados por classes derivadas da classe.
público	Membros de classe declarados como public podem ser usados por qualquer função.

O controle de acesso ajuda a evitar que você use objetos de maneiras que não sejam as destinadas a eles. Essa proteção será perdida quando as conversões de tipo explícitas (casts) forem executadas.

NOTE

O controle de acesso é igualmente aplicável a todos os nomes: funções membro, dados de membro, classes aninhadas e enumeradores.

Controle de acesso em classes derivadas

Dois fatores controlam quais membros de uma classe base são acessíveis em uma classe derivada; esses mesmos fatores controlam o acesso aos membros herdados na classe derivada:

- Se a classe derivada declara a classe base usando o `public` especificador de acesso.
- Qual é o acesso ao membro na classe base.

A tabela a seguir mostra a interação entre esses fatores e como determinar o acesso do membro da classe base.

Acesso do membro na classe base

PARTICULARES	PROTEGIDOS	PÚBLICO
Sempre inacessível independentemente do acesso de derivação	Privado na classe derivada se você usar derivação particular	Privado na classe derivada se você usar derivação particular
	Protegido na classe derivada se você usar derivação protegida	Protegido na classe derivada se você usar derivação protegida
	Protegido na classe derivada se você usar derivação pública	Público na classe derivada se você usar derivação pública

O exemplo a seguir ilustra isso:

```

// access_specifiers_for_base_classes.cpp
class BaseClass
{
public:
    int PublicFunc(); // Declare a public member.
protected:
    int ProtectedFunc(); // Declare a protected member.
private:
    int PrivateFunc(); // Declare a private member.
};

// Declare two classes derived from BaseClass.
class DerivedClass1 : public BaseClass
{
    void foo()
    {
        PublicFunc();
        ProtectedFunc();
        PrivateFunc(); // function is inaccessible
    }
};

class DerivedClass2 : private BaseClass
{
    void foo()
    {
        PublicFunc();
        ProtectedFunc();
        PrivateFunc(); // function is inaccessible
    }
};

int main()
{
    DerivedClass1 derived_class1;
    DerivedClass2 derived_class2;
    derived_class1.PublicFunc();
    derived_class2.PublicFunc(); // function is inaccessible
}

```

Em `DerivedClass1`, a função de membro `PublicFunc` é um membro público e `ProtectedFunc` é um membro protegido porque `BaseClass` é uma classe base pública. `PrivateFunc` é particular a `BaseClass`, e inacessível a quaisquer classes derivadas.

Em `DerivedClass2`, as funções `PublicFunc` e `ProtectedFunc` são consideradas membros particulares porque `BaseClass` é uma classe base particular. Novamente, `PrivateFunc` é particular a `BaseClass`, e inacessível a quaisquer classes derivadas.

Você pode declarar uma classe derivada sem um especificador de acesso de classe base. Nesse caso, a derivação será considerada privada se a declaração de classe derivada usar a `class` palavra-chave. A derivação será considerada pública se a declaração de classe derivada usar a `struct` palavra-chave. Por exemplo, o código a seguir:

```

class Derived : Base
...

```

é equivalente a:

```

class Derived : private Base
...

```

Da mesma forma, o código a seguir:

```
struct Derived : Base  
{  
    ...  
}
```

é equivalente a:

```
struct Derived : public Base  
{  
    ...
```

Observe que os membros declarados como tendo acesso privado não são acessíveis a funções ou classes derivadas, a menos que essas funções ou classes sejam declaradas usando a `friend` declaração na classe base.

Um `union` tipo não pode ter uma classe base.

NOTE

Ao especificar uma classe base privada, é recomendável usar explicitamente a `private` palavra-chave para que os usuários da classe derivada compreendam o acesso do membro.

Controle de acesso e membros estáticos

Quando você especifica uma classe base como `private`, ela afeta somente membros não estáticos. Os membros estáticos públicos ainda são acessíveis nas classes derivadas. No entanto, o acesso a membros da classe base usando ponteiros, referências ou objetos pode exigir uma conversão, quando então o controle de acesso é aplicado novamente. Considere o exemplo a seguir:

No código acima, o controle de acesso proíbe a conversão de um ponteiro para `Derived2` em um ponteiro para `Base`. O `this` ponteiro é implicitamente do tipo `Derived2 *`. Para selecionar a `Countof` função, `this` é necessário convertê-la no tipo `Base *`. Essa conversão não é permitida porque `Base` é uma classe base indireta privada para `Derived2`. A conversão em um tipo de classe base privada é aceitável apenas no caso de ponteiros para as classes derivadas imediatas. Consequentemente, os ponteiros do tipo `Derived1 *` podem ser convertidos no tipo `Base *`.

Observe que chamar a função `Countof` explicitamente, sem usar um ponteiro, uma referência ou um objeto para selecioná-la, não implica em nenhuma conversão. Consequentemente, a chamada é permitida.

Os membros e amigos de uma classe derivada, `T`, podem converter um ponteiro para `T` em um ponteiro para uma classe base direta privada de `T`.

Acesso a funções virtuais

O controle de acesso aplicado às funções [virtuais](#) é determinado pelo tipo usado para fazer a chamada de função. Substituir declarações da função não afeta o controle de acesso para um determinado tipo. Por exemplo:

```
// access_to_virtual_functions.cpp
class VFuncBase
{
public:
    virtual int GetState() { return _state; }
protected:
    int _state;
};

class VFuncDerived : public VFuncBase
{
private:
    int GetState() { return _state; }
};

int main()
{
    VFuncDerived vfd;           // Object of derived type.
    VFuncBase *pvfb = &vfd;    // Pointer to base type.
    VFuncDerived *pvfd = &vfd;  // Pointer to derived type.
    int State;

    State = pvfb->GetState(); // GetState is public.
    State = pvfd->GetState(); // C2248 error expected; GetState is private;
}
```

No exemplo anterior, chamar a função virtual `GetState` usando um ponteiro para o tipo `VFuncBase` chama `VFuncDerived::GetState`, e `GetState` é tratada como pública. No entanto, chamar `GetState` usando um ponteiro para o tipo `VFuncDerived` é uma violação de controle de acesso porque `GetState` é declarado como particular na classe `VFuncDerived`.

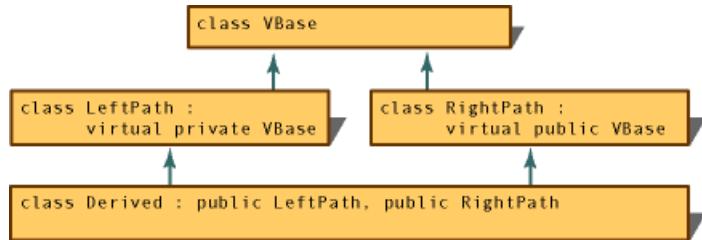
Caution

A função virtual `GetState` pode ser chamada usando um ponteiro para a classe base `VFuncBase`. Isso não significa que a função chamada seja a versão da classe base dessa função.

Controle de acesso com várias heranças

Em células entrelaçadas de herança múltipla que envolvem classes base virtuais, um nome determinado pode ser acessado por mais de um caminho. Como o controle de acesso diferente pode ser aplicado ao longo desses caminhos diferentes, o compilador escolhe o caminho que fornece o maior acesso. Consulte a

figura a seguir.



Acessar nos caminhos de um grafo de herança

Na figura, um nome declarado na classe `VBase` é sempre acessado pela classe `RightPath`. O caminho à direita é mais acessível porque `RightPath` declara `VBase` como uma classe base pública, enquanto que `LeftPath` declara `VBase` como particular.

Confira também

[Referência da linguagem C++](#)

friend (C++)

02/09/2020 • 10 minutes to read • [Edit Online](#)

Em algumas circunstâncias, é mais conveniente conceder acesso em nível de membro a funções que não são membros de uma classe ou a todos os membros em uma classe separada. Somente o implementador de classe pode declarar quem são seus amigos. Uma função ou classe não pode se declarar como um amigo de qualquer classe. Em uma definição de classe, use a `friend` palavra-chave e o nome de uma função não membro ou outra classe para conceder a ela acesso aos membros privados e protegidos da sua classe. Em uma definição de modelo, um parâmetro de tipo pode ser declarado como um amigo.

Sintaxe

```
class friend F
friend F;
```

Declarações Friend

Se você declarar uma função de amigo que não foi declarada anteriormente, essa função é exportada para o escopo de envolvimento sem classe.

As funções declaradas em uma declaração Friend são tratadas como se tivessem sido declaradas usando a `extern` palavra-chave. Para obter mais informações, consulte [extern](#).

Ainda que as funções com escopo global possam ser declaradas como amigos antes dos protótipos, as funções de membro não podem ser declaradas como amigos antes da aparência de sua declaração completa da classe. O exemplo de código a seguir mostra porque isso falha:

```
class ForwardDeclared; // Class name is known.
class HasFriends
{
    friend int ForwardDeclared::IsAFriend(); // C2039 error expected
};
```

O exemplo anterior insere o nome da classe `ForwardDeclared` no escopo, mas a declaração completa — especificamente, a parte que declara a função `IsAFriend` — não é conhecida. Portanto, a `friend` declaração na classe `HasFriends` gera um erro.

A partir do C++ 11, há duas formas de declarações Friend para uma classe:

```
friend class F;
friend F;
```

O primeiro formulário apresenta uma nova classe F se nenhuma classe existente com esse nome foi encontrada no namespace mais interno. C++ 11: o segundo formulário não introduz uma nova classe; Ele pode ser usado quando a classe já foi declarada e deve ser usada ao declarar um parâmetro de tipo de modelo ou um `typedef` como um Friend.

Use `class friend F` quando o tipo referenciado ainda não tiver sido declarado:

```
namespace NS
{
    class M
    {
        class friend F; // Introduces F but doesn't define it
    };
}
```

```
namespace NS
{
    class M
    {
        friend F; // error C2433: 'NS::F': 'friend' not permitted on data declarations
    };
}
```

No exemplo a seguir, `friend F` refere-se à `F` classe declarada fora do escopo do NS.

```
class F {};
namespace NS
{
    class M
    {
        friend F; // OK
    };
}
```

Use `friend F` para declarar um parâmetro de modelo como um amigo:

```
template <typename T>
class my_class
{
    friend T;
    //...
};
```

Use `friend F` para declarar um `typedef` como `Friend`:

```
class Foo {};
typedef Foo F;

class G
{
    friend F; // OK
    friend class F // Error C2371 -- redefinition
};
```

Para declarar duas classes que são amigas da outro, a segunda classe inteira deve ser especificada como amiga de primeira classe. A razão para essa restrição é que o compilador tem informações suficientes para declarar funções individuais de amigo somente no ponto onde a segunda classe é declarada.

NOTE

Embora a segunda classe inteira deve ser amiga da primeira classe, você pode selecionar quais funções na primeira classe serão amigas da segunda classe.

funções friend

Uma `friend` função é uma função que não é um membro de uma classe, mas tem acesso aos membros privados e protegidos da classe. As funções friend não são consideradas membros de classe; elas são funções externas normais que recebem privilégios de acesso especiais. Os amigos não estão no escopo da classe e não são chamados usando os operadores de seleção de Membros (. and ->) a menos que sejam membros de outra classe. Uma `friend` função é declarada pela classe que está concedendo acesso. A `friend` declaração pode ser colocada em qualquer lugar na declaração de classe. Ela não é afetada pelas palavras-chave de controle de acesso.

O exemplo a seguir mostra uma classe `Point` e uma função friend, `ChangePrivate`. A `friend` função tem acesso ao membro de dados privados do `Point` objeto que recebe como um parâmetro.

```
// friend_functions.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
class Point
{
    friend void ChangePrivate( Point & );
public:
    Point( void ) : m_i(0) {}
    void PrintPrivate( void ){cout << m_i << endl; }

private:
    int m_i;
};

void ChangePrivate ( Point &i ) { i.m_i++; }

int main()
{
    Point sPoint;
    sPoint.PrintPrivate();
    ChangePrivate(sPoint);
    sPoint.PrintPrivate();
    // Output: 0
    //          1
}
```

Membros da classe como amigos

As funções de membro da classe podem ser declaradas como amigos em outras classes. Considere o exemplo a seguir:

```

// classes_as_friends1.cpp
// compile with: /c
class B;

class A {
public:
    int Func1( B& b );

private:
    int Func2( B& b );
};

class B {
private:
    int _b;

    // A::Func1 is a friend function to class B
    // so A::Func1 has access to all members of B
    friend int A::Func1( B& );
};

int A::Func1( B& b ) { return b._b; }    // OK
int A::Func2( B& b ) { return b._b; }    // C2248

```

No exemplo anterior, somente a função `A::Func1(B&)` tem acesso de amigo para classificar `B`. Portanto, o acesso ao membro privado `_b` está correto na `Func1` classe `A`, mas não no `Func2`.

Uma `friend` classe é uma classe que todas as funções de membro são funções Friend de uma classe, ou seja, cujas funções de membro têm acesso aos membros privados e protegidos da outra classe. Suponha que a `friend` declaração na classe `B` tenha sido:

```
friend class A;
```

Nesse caso, todas as funções membro na classe `A` receberão acesso de amigo para classificar `B`. O código a seguir é um exemplo de uma classe de amigo:

```

// classes_as_friends2.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
class YourClass {
friend class YourOtherClass; // Declare a friend class
public:
    YourClass() : topSecret(0){}
    void printMember() { cout << topSecret << endl; }
private:
    int topSecret;
};

class YourOtherClass {
public:
    void change( YourClass& yc, int x ){yc.topSecret = x;}
};

int main() {
    YourClass yc1;
    YourOtherClass yoc1;
    yc1.printMember();
    yoc1.change( yc1, 5 );
    yc1.printMember();
}

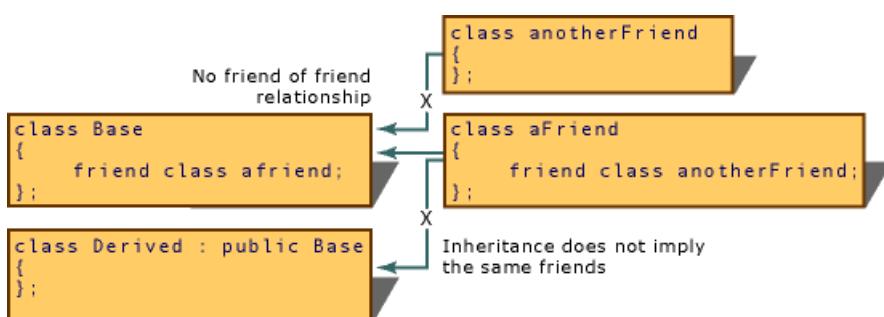
```

A amizade não é mútua a menos que explicitamente especificada como tal. No exemplo anterior, as funções membro de `YourClass` não podem acessar os membros privados de `YourOtherClass`.

Um tipo gerenciado (em C++/CLI) não pode ter nenhuma função Friend, Friend classes ou Friend interfaces.

A amizade não é herdada, o que significa que as classes derivadas de `YourOtherClass` não podem acessar os membros privados de `YourClass`. A amizade não é transitiva, assim as classes que são amigos de `YourOtherClass` não podem acessar membros privados de `YourClass`.

A figura a seguir mostra quatro declarações de classe: `Base`, `Derived`, `aFriend` e `anotherFriend`. Somente a classe `aFriend` tem acesso direto aos membros privados de `Base` (e a todos os membros que `Base` pode ter herdado).



Implicações da relação Friend

Definições de amigo embutido

As funções Friend podem ser definidas (dadas um corpo de função) dentro de declarações de classe. Essas funções são funções embutidas, e como funções membro embutidas, se comportam como se fossem definidas imediatamente após todos os membros de classe terem sido vistos, mas antes que o escopo de classe seja fechado (o final da declaração de classe). Funções Friend definidas dentro de declarações de classe estão no escopo da classe delimitadora.

Confira também

[Palavras-chave](#)

private (C++)

02/09/2020 • 2 minutes to read • [Edit Online](#)

Sintaxe

```
private:  
  [member-list]  
private base-class
```

Comentários

Ao anteceder uma lista de membros de classe, a `private` palavra-chave especifica que esses membros são acessíveis somente de funções de membro e amigos da classe. Isso se aplica a todos os membros declarados até o especificador seguinte do acesso ou ao final da classe.

Ao anteceder o nome de uma classe base, a `private` palavra-chave especifica que os membros públicos e protegidos da classe base são membros privados da classe derivada.

O acesso padrão dos membros em um classe é particular. O acesso padrão dos membros em uma estrutura ou união é público.

O acesso padrão de uma classe base é particular para classes e público para estruturas. Uniões não podem ter classes base.

Para obter informações relacionadas, consulte [amigo](#), [público](#), [protegido](#) e a tabela de acesso a membros no [controle de acesso a membros de classe](#).

Específico do /clr

Em tipos CLR, as palavras-chave do especificador de acesso C++ (`public`, `private` e `protected`) podem afetar a visibilidade de tipos e métodos em relação a assemblies. Para obter mais informações, consulte [controle de acesso de membro](#).

NOTE

Arquivos compilados com `/In` não são afetados por esse comportamento. Nesse caso, todas as classes gerenciadas (públicas ou particulares) serão visíveis.

Específico de END /clr

Exemplo

```
// keyword_private.cpp
class BaseClass {
public:
    // privMem accessible from member function
    int pubFunc() { return privMem; }
private:
    void privMem;
};

class DerivedClass : public BaseClass {
public:
    void usePrivate( int i )
    { privMem = i; }    // C2248: privMem not accessible
                        // from derived class
};

class DerivedClass2 : private BaseClass {
public:
    // pubFunc() accessible from derived class
    int usePublic() { return pubFunc(); }
};

int main() {
    BaseClass aBase;
    DerivedClass aDerived;
    DerivedClass2 aDerived2;
    aBase.privMem = 1;      // C2248: privMem not accessible
    aDerived.privMem = 1;   // C2248: privMem not accessible
                           // in derived class
    aDerived2.pubFunc();   // C2247: pubFunc() is private in
                           // derived class
}
```

Confira também

[Controlando o acesso a membros de classe](#)

[Palavras-chave](#)

protected (C++)

02/09/2020 • 3 minutes to read • [Edit Online](#)

Sintaxe

```
protected:  
  [member-list]  
protected base-class
```

Comentários

A `protected` palavra-chave especifica o acesso a membros de classe na *lista de membros* até o próximo especificador de acesso (`public` ou `private`) ou o final da definição de classe. Membros de classe declarados como `protected` podem ser usados somente pelo seguinte:

- Funções de membro da classe que declarou originalmente esses membros.
- Friends da classe que declarou originalmente esses membros.
- Classes derivadas com acesso público ou protegido da classe que declarou originalmente esses membros.
- Direcionar classes derivadas de modo particular que também têm acesso aos membros protegidos.

Ao anteceder o nome de uma classe base, a `protected` palavra-chave especifica que os membros públicos e protegidos da classe base são membros protegidos de suas classes derivadas.

Membros protegidos não são tão privados como `private` Membros, que são acessíveis somente para membros da classe na qual eles são declarados, mas não são tão públicos como `public` Membros, que podem ser acessados em qualquer função.

Membros protegidos que também são declarados como `static` estão acessíveis a qualquer função Friend ou membro de uma classe derivada. Membros protegidos que não são declarados como `static` estão acessíveis a amigos e funções de membro em uma classe derivada somente por meio de um ponteiro para, referência a ou objeto da classe derivada.

Para obter informações relacionadas, consulte [amigo](#), [público](#), [privado](#) e a tabela de acesso a membros no [controle de acesso a membros de classe](#).

Específico do /clr

Em tipos CLR, as palavras-chave do especificador de acesso C++ (`public`, `private` e `protected`) podem afetar a visibilidade de tipos e métodos em relação a assemblies. Para obter mais informações, consulte [controle de acesso de membro](#).

NOTE

Arquivos compilados com `/In` não são afetados por esse comportamento. Nesse caso, todas as classes gerenciadas (públicas ou particulares) serão visíveis.

Específico de END /clr

Exemplo

```
// keyword_protected.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
class X {
public:
    void setProtMemb( int i ) { m_protMemb = i; }
    void Display() { cout << m_protMemb << endl; }
protected:
    int m_protMemb;
    void Protfunc() { cout << "\nAccess allowed\n"; }
} x;

class Y : public X {
public:
    void useProtfunc() { Protfunc(); }
} y;

int main() {
    // x.m_protMemb;           error, m_protMemb is protected
    x.setProtMemb( 0 );      // OK, uses public access function
    x.Display();
    y.setProtMemb( 5 );      // OK, uses public access function
    y.Display();
    // x.Protfunc();          error, Protfunc() is protected
    y.useProtfunc();          // OK, uses public access function
                             // in derived class
}
```

Confira também

[Controlando o acesso a membros de classe](#)

[Palavras-chave](#)

public (C++)

02/09/2020 • 2 minutes to read • [Edit Online](#)

Sintaxe

```
public:  
  [member-list]  
public base-class
```

Comentários

Ao anteceder uma lista de membros de classe, a `public` palavra-chave especifica que esses membros podem ser acessados de qualquer função. Isso se aplica a todos os membros declarados até o especificador seguinte do acesso ou ao final da classe.

Ao anteceder o nome de uma classe base, a `public` palavra-chave especifica que os membros públicos e protegidos da classe base são membros públicos e protegidos, respectivamente, da classe derivada.

O acesso padrão dos membros em um classe é particular. O acesso padrão dos membros em uma estrutura ou união é público.

O acesso padrão de uma classe base é particular para classes e público para estruturas. Uniões não podem ter classes base.

Para obter mais informações, consulte [privado](#), [protegido](#), [amigo](#) e a tabela de acesso a membro no [controle de acesso a membros de classe](#).

Específico do /clr

Em tipos CLR, as palavras-chave do especificador de acesso C++ (`public`, `private` e `protected`) podem afetar a visibilidade de tipos e métodos em relação a assemblies. Para obter mais informações, consulte [controle de acesso de membro](#).

NOTE

Arquivos compilados com `/In` não são afetados por esse comportamento. Nesse caso, todas as classes gerenciadas (públicas ou particulares) serão visíveis.

Específico de END /clr

Exemplo

```
// keyword_public.cpp
class BaseClass {
public:
    int pubFunc() { return 0; }
};

class DerivedClass : public BaseClass {};

int main() {
    BaseClass aBase;
    DerivedClass aDerived;
    aBase.pubFunc();           // pubFunc() is accessible
                             // from any function
    aDerived.pubFunc();        // pubFunc() is still public in
                             // derived class
}
```

Confira também

[Controlando o acesso a membros de classe](#)

[Palavras-chave](#)

Inicialização de chaves

02/09/2020 • 5 minutes to read • [Edit Online](#)

Nem sempre é necessário definir um construtor para uma classe, especialmente aqueles que são relativamente simples. Os usuários podem inicializar objetos de uma classe ou estrutura usando a inicialização uniforme, conforme mostrado no exemplo a seguir:

```
// no_constructor.cpp
// Compile with: cl /EHsc no_constructor.cpp
#include <time.h>

// No constructor
struct TempData
{
    int StationId;
    time_t timeSet;
    double current;
    double maxTemp;
    double minTemp;
};

// Has a constructor
struct TempData2
{
    TempData2(double minimum, double maximum, double cur, int id, time_t t) :
        stationId{id}, timeSet{t}, current{cur}, maxTemp{maximum}, minTemp{minimum} {}
    int stationId;
    time_t timeSet;
    double current;
    double maxTemp;
    double minTemp;
};

int main()
{
    time_t time_to_set;

    // Member initialization (in order of declaration):
    TempData td{ 45978, time(&time_to_set), 28.9, 37.0, 16.7 };

    // Default initialization = {0,0,0,0,0}
    TempData td_default{};

    // Uninitialized = if used, emits warning C4700 uninitialized local variable
    TempData td_noInit;

    // Member declaration (in order of ctor parameters)
    TempData2 td2{ 16.7, 37.0, 28.9, 45978, time(&time_to_set) };

    return 0;
}
```

Observe que, quando uma classe ou struct não tem nenhum construtor, você fornece os elementos da lista na ordem em que os membros são declarados na classe. Se a classe tiver um construtor, forneça os elementos na ordem dos parâmetros. Se um tipo tiver um construtor padrão, implicitamente ou explicitamente declarado, você poderá usar a inicialização de chave padrão (com chaves vazias). Por exemplo, a seguinte classe pode ser inicializada usando a inicialização de chave padrão e não padrão:

```

#include <string>
using namespace std;

class class_a {
public:
    class_a() {}
    class_a(string str) : m_string{ str } {}
    class_a(string str, double dbl) : m_string{ str }, m_double{ dbl } {}
    double m_double;
    string m_string;
};

int main()
{
    class_a c1{};
    class_a c1_1;

    class_a c2{ "ww" };
    class_a c2_1("xx");

    // order of parameters is the same as the constructor
    class_a c3{ "yy", 4.4 };
    class_a c3_1("zz", 5.5);
}

```

Se uma classe tiver construtores não padrão, a ordem na qual os membros da classe aparecem no inicializador de chave é a ordem na qual os parâmetros correspondentes aparecem no construtor, não a ordem na qual os membros são declarados (como `class_a` no exemplo anterior). Caso contrário, se o tipo não tiver um construtor declarado, a ordem na qual os membros aparecerão no inicializador de chaves será igual à ordem na qual eles são declarados; Nesse caso, você pode inicializar quantos membros públicos desejar, mas não pode ignorar nenhum membro. O exemplo a seguir mostra a ordem usada na inicialização de chaves quando não há um construtor declarado:

```

class class_d {
public:
    float m_float;
    string m_string;
    wchar_t m_char;
};

int main()
{
    class_d d1{};
    class_d d1{ 4.5 };
    class_d d2{ 4.5, "string" };
    class_d d3{ 4.5, "string", 'c' };

    class_d d4{ "string", 'c' }; // compiler error
    class_d d5{ "string", 'c', 2.0 }; // compiler error
}

```

Se o construtor padrão for declarado explicitamente, mas marcado como excluído, a inicialização de chave padrão não poderá ser usada:

```

class class_f {
public:
    class_f() = delete;
    class_f(string x): m_string { x } {}
    string m_string;
};

int main()
{
    class_f cf{ "hello" };
    class_f cf1{}; // compiler error C2280: attempting to reference a deleted function
}

```

Você pode usar a inicialização de chaves em qualquer lugar em que normalmente faria a inicialização — por exemplo, como um parâmetro de função ou um valor de retorno, ou com a `new` palavra-chave:

```

class_d* cf = new class_d{4.5};
kr->add_d({ 4.5 });
return { 4.5 };

```

No modo `/std: c++ 17`, as regras para a inicialização de chaves vazias são um pouco mais restritivas. Consulte [construtores derivados e inicialização de agregação estendida](#).

construtores de initializer_list

A [classe initializer_list](#) representa uma lista de objetos de um tipo especificado que podem ser usados em um construtor e em outros contextos. Você pode construir um `initializer_list` usando a inicialização de chaves:

```
initializer_list<int> int_list{5, 6, 7};
```

IMPORTANT

Para usar essa classe, você deve incluir o `<initializer_list>` cabeçalho.

Um `initializer_list` pode ser copiado. Nesse caso, os membros da nova lista são referências aos membros da lista original:

```

initializer_list<int> ilist1{ 5, 6, 7 };
initializer_list<int> ilist2( ilist1 );
if (ilist1.begin() == ilist2.begin())
    cout << "yes" << endl; // expect "yes"

```

As classes de contêiner de biblioteca padrão, e também, `string`, `wstring` e `regex` têm `initializer_list` construtores. Os exemplos a seguir mostram como fazer a inicialização de chaves com estes construtores:

```

vector<int> v1{ 9, 10, 11 };
map<int, string> m1{ {1, "a"}, {2, "b"} };
string s{ 'a', 'b', 'c' };
regex rgx{ 'x', 'y', 'z' };

```

Confira também

[Classes e structs](#)

[Construtores](#)

Tempo de vida do objeto e gerenciamento de recursos (RAII)

21/03/2020 • 5 minutes to read • [Edit Online](#)

Diferentemente das linguagens gerenciadas, C++ não tem coleta de lixo automática. Esse é um processo interno que libera a memória de heap e outros recursos à medida que um programa é executado. Um C++ programa é responsável por retornar todos os recursos adquiridos para o sistema operacional. Falha ao liberar um recurso não utilizado é chamado de *vazamento*. Os recursos vazados ficam indisponíveis para outros programas até que o processo seja encerrado. Vazamentos de memória em particular são uma causa comum de bugs na programação em estilo C.

O C++ moderno evita o uso da memória heap o máximo possível declarando objetos na pilha. Quando um recurso é muito grande para a pilha, ele deve *pertencer* a um objeto. À medida que o objeto é inicializado, ele adquire o recurso que ele possui. O objeto é então responsável por liberar o recurso em seu destruidor. O próprio objeto proprietário é declarado na pilha. O princípio que os *próprios recursos de objetos* também é conhecido como "a aquisição de recursos é inicialização" ou RAII.

Quando um objeto de pilha de Propriedade do recurso sai do escopo, seu destruidor é invocado automaticamente. Dessa forma, a coleta de lixo C++ no está bem relacionada à vida útil do objeto e é determinística. Um recurso é sempre liberado em um ponto conhecido no programa, que você pode controlar. Somente destruidores determinísticos como aqueles C++ no podem manipular memória e recursos sem memória igualmente.

O exemplo a seguir mostra um objeto simples `w`. Ele é declarado na pilha no escopo da função e é destruído no final do bloco de função. O objeto `w` não possui *recursos* (como memória alocada para heap). Seu único membro `g` é declarado na pilha e simplesmente sai do escopo junto com `w`. Nenhum código especial é necessário no destruidor `widget`.

```
class widget {
private:
    gadget g;    // lifetime automatically tied to enclosing object
public:
    void draw();
};

void functionUsingWidget () {
    widget w;    // lifetime automatically tied to enclosing scope
                 // constructs w, including the w.g gadget member
    // ...
    w.draw();
    // ...
} // automatic destruction and deallocation for w and w.g
  // automatic exception safety,
  // as if "finally { w.dispose(); w.g.dispose(); }"
```

No exemplo a seguir, `w` possui um recurso de memória e, portanto, deve ter código em seu destruidor para excluir a memória.

```

class widget
{
private:
    int* data;
public:
    widget(const int size) { data = new int[size]; } // acquire
    ~widget() { delete[] data; } // release
    void do_something() {}
};

void functionUsingWidget() {
    widget w(1000000);    // lifetime automatically tied to enclosing scope
                          // constructs w, including the w.data member
    w.do_something();

} // automatic destruction and deallocation for w and w.data

```

Desde o C++ 11, há uma maneira melhor de escrever o exemplo anterior: usando um ponteiro inteligente da biblioteca padrão. O ponteiro inteligente manipula a alocação e a exclusão da memória que ela possui. O uso de um ponteiro inteligente elimina a necessidade de um destruidor explícito na classe `widget`.

```

#include <memory>
class widget
{
private:
    std::unique_ptr<int> data;
public:
    widget(const int size) { data = std::make_unique<int>(size); }
    void do_something() {}
};

void functionUsingWidget() {
    widget w(1000000);    // lifetime automatically tied to enclosing scope
                          // constructs w, including the w.data gadget member
    // ...
    w.do_something();
    // ...
} // automatic destruction and deallocation for w and w.data

```

Usando ponteiros inteligentes para alocação de memória, você pode eliminar o potencial de vazamentos de memória. Esse modelo funciona para outros recursos, como identificadores de arquivo ou soquetes. Você pode gerenciar seus próprios recursos de maneira semelhante em suas classes. Para obter mais informações, consulte [ponteiros inteligentes](#).

O design de C++ garante que os objetos sejam destruídos quando saem do escopo. Ou seja, eles são destruídos conforme os blocos são encerrados, na ordem inversa de construção. Quando um objeto é destruído, suas bases e membros são destruídos em uma ordem específica. Objetos declarados fora de qualquer bloco, no escopo global, podem causar problemas. Pode ser difícil depurar, se o construtor de um objeto global lançar uma exceção.

Confira também

[Bem-vindo de volta para C++](#)

[Referência da linguagem C++](#)

[Biblioteca padrão C++](#)

Pimpl para encapsulamento do tempo de compilação (C++ moderno)

02/12/2019 • 2 minutes to read • [Edit Online](#)

O *idioma pimpl* é uma técnica C++ moderna para ocultar a implementação, para minimizar o acoplamento e para separar interfaces. Pimpl é curto para "ponteiro para implementação". Talvez você já esteja familiarizado com o conceito, mas conheça-o por outros nomes, como Cheshire Cat ou idioma do firewall do compilador.

Por que usar o pimpl?

Veja como o idioma pimpl pode melhorar o ciclo de vida do desenvolvimento de software:

- Minimização de dependências de compilação.
- Separação de interface e implementação.
- Portabilidade.

Cabeçalho Pimpl

```
// my_class.h
class my_class {
    // ... all public and protected stuff goes here ...
private:
    class impl; unique_ptr<impl> pimpl; // opaque type here
};
```

O idioma pimpl evita a recompilação em cascata e layouts de objeto frágil. Ele é adequado para tipos populares (transitivamente).

Implementação de Pimpl

Defina a classe `impl` no arquivo `.cpp`.

```
// my_class.cpp
class my_class::impl { // defined privately here
    // ... all private data and functions: all of these
    //      can now change without recompiling callers ...
};

my_class::my_class(): pimpl( new impl )
{
    // ... set impl values ...
}
```

Práticas recomendadas

Considere se é para adicionar suporte para especialização de permuta sem lançamento.

Consulte também

[Bem-vindo de volta para C++](#)

Referência da linguagem C++

Biblioteca padrão C++

Portabilidade nos limites da ABI

02/12/2019 • 2 minutes to read • [Edit Online](#)

Use tipos e convenções suficientemente portáteis em limites de interface binários. Um "tipo portátil" é um tipo interno C ou uma struct que contém apenas tipos internos de C. Os tipos de classe só podem ser usados quando o chamador e o receptor concordem no layout, na Convenção de chamada, etc. Isso só é possível quando ambos são compilados com as mesmas configurações de compilador e compilador.

Como mesclar uma classe para portabilidade C

Quando os chamadores podem ser compilados com outro compilador/linguagem, então "achatar" para uma API "C" **externa** com uma Convenção de chamada específica:

```
// class widget {
//     widget();
//     ~widget();
//     double method( int, gadget& );
// };
extern "C" {           // functions using explicit "this"
    struct widget;    // opaque type (forward declaration only)
    widget* STDCALL widget_create();        // constructor creates new "this"
    void STDCALL widget_destroy(widget*); // destructor consumes "this"
    double STDCALL widget_method(widget*, int, gadget*); // method uses "this"
}
```

Consulte também

[Bem-vindo de volta paraC++](#)

[Referência da linguagem C++](#)

[Biblioteca padrão C++](#)

Construtores (C++)

02/09/2020 • 30 minutes to read • [Edit Online](#)

Para personalizar como os membros de classe são inicializados ou para invocar funções quando um objeto da sua classe é criado, defina um *Construtor*. Um construtor tem o mesmo nome que a classe e nenhum valor de retorno. Você pode definir quantos construtores sobrecarregados forem necessários para personalizar a inicialização de várias maneiras. Normalmente, os construtores têm acessibilidade pública para que o código fora da definição de classe ou da hierarquia de herança possa criar objetos da classe. Mas você também pode declarar um construtor como `protected` ou `private`.

Os construtores podem, opcionalmente, usar uma lista inicial de membros. Essa é uma maneira mais eficiente de inicializar membros de classe do que atribuir valores no corpo do construtor. O exemplo a seguir mostra uma classe `Box` com três construtores sobrecarregados. As duas últimas usam listas de iniciais de membros:

```
class Box {
public:
    // Default constructor
    Box() {}

    // Initialize a Box with equal dimensions (i.e. a cube)
    explicit Box(int i) : m_width(i), m_length(i), m_height(i) // member init list
    {}

    // Initialize a Box with custom dimensions
    Box(int width, int length, int height)
        : m_width(width), m_length(length), m_height(height)
    {}

    int Volume() { return m_width * m_length * m_height; }

private:
    // Will have value of 0 when default constructor is called.
    // If we didn't zero-init here, default constructor would
    // leave them uninitialized with garbage values.
    int m_width{ 0 };
    int m_length{ 0 };
    int m_height{ 0 };
};
```

Quando você declara uma instância de uma classe, o compilador escolhe qual Construtor invocar com base nas regras de resolução de sobrecarga:

```
int main()
{
    Box b; // Calls Box()

    // Using uniform initialization (preferred):
    Box b2 {5}; // Calls Box(int)
    Box b3 {5, 8, 12}; // Calls Box(int, int, int)

    // Using function-style notation:
    Box b4(2, 4, 6); // Calls Box(int, int, int)
}
```

- Os construtores podem ser declarados como `inline`, `explícitos` `friend` ou `constexpr`.
- Um construtor pode inicializar um objeto que foi declarado como `const` `volatile` ou `const volatile`. O

objeto se torna `const` após a conclusão do construtor.

- Para definir um construtor em um arquivo de implementação, dê a ele um nome qualificado como com qualquer outra função de membro: `Box::Box(){...}` .

Listas de inicializadores de membros

Um construtor, opcionalmente, pode ter uma lista de inicializadores de membros, que Inicializa membros de classe antes da execução do corpo do construtor. (Observe que uma lista de inicializadores de membros não é a mesma coisa que uma *lista de inicializadores* do tipo `std:: initializer_list`.)

O uso de uma lista de inicializadores de membros é preferível ao atribuir valores no corpo do Construtor, pois inicializa diretamente o membro. No exemplo a seguir, mostra que a lista de inicializadores de membro consiste em todas as expressões de **identificador (argumento)** após os dois-pontos:

```
Box(int width, int length, int height)
    : m_width(width), m_length(length), m_height(height)
{}
```

O identificador deve se referir a um membro de classe; Ele é inicializado com o valor do argumento. O argumento pode ser um dos parâmetros do Construtor, uma chamada de função ou um `std:: initializer_list`.

`const` Membros e membros do tipo de referência devem ser inicializados na lista de inicializadores de membro.

As chamadas para construtores de classe base com parâmetros devem ser feitas na lista de inicializadores para garantir que a classe base seja totalmente inicializada antes da execução do Construtor derivado.

Construtores padrão

Os construtores padrão normalmente não têm parâmetros, mas podem ter parâmetros com valores padrão.

```
class Box {
public:
    Box() { /*perform any required default initialization steps*/}

    // All params have default values
    Box (int w = 1, int l = 1, int h = 1): m_width(w), m_height(h), m_length(l){}
    ...
}
```

Os construtores padrão são uma das **funções de membro especiais**. Se nenhum construtor for declarado em uma classe, o compilador fornecerá um `inline` construtor padrão implícito.

```

#include <iostream>
using namespace std;

class Box {
public:
    int Volume() {return m_width * m_height * m_length;}
private:
    int m_width { 0 };
    int m_height { 0 };
    int m_length { 0 };
};

int main() {
    Box box1; // Invoke compiler-generated constructor
    cout << "box1.Volume: " << box1.Volume() << endl; // Outputs 0
}

```

Se você depender de um construtor padrão implícito, certifique-se de inicializar os membros na definição de classe, conforme mostrado no exemplo anterior. Sem esses inicializadores, os membros não são inicializados e a chamada de volume () produziria um valor de lixo. Em geral, é uma boa prática inicializar membros dessa maneira mesmo quando não depender de um construtor padrão implícito.

Você pode impedir que o compilador gere um construtor padrão implícito definindo-o como [excluído](#):

```

// Default constructor
Box() = delete;

```

Um construtor padrão gerado pelo compilador será definido como excluído se qualquer membro de classe não for default-constructible. Por exemplo, todos os membros do tipo de classe e seus membros de tipo de classe, devem ter um construtor padrão e destruidores acessíveis. Todos os membros de dados do tipo de referência, bem como `const` os membros, devem ter um inicializador de membro padrão.

Quando você chama um construtor padrão gerado pelo compilador e tenta usar parênteses, um aviso é emitido:

```

class myclass{};
int main(){
    myclass mc();      // warning C4930: prototyped function not called (was a variable definition intended?)
}

```

Este é um exemplo do problema Most Vexing Parse. Como a expressão de exemplo pode ser interpretada como a declaração de uma função ou como invocação de um construtor padrão, e como os analisadores de C++ preferem as declarações a outras coisas, a expressão é tratada como uma declaração de função. Para obter mais informações, consulte [análise mais irritante](#).

Se algum construtor não padrão for declarado, o compilador não fornecerá um construtor padrão:

```

class Box {
public:
    Box(int width, int length, int height)
        : m_width(width), m_length(length), m_height(height){}
private:
    int m_width;
    int m_length;
    int m_height;
};

int main(){

    Box box1(1, 2, 3);
    Box box2{ 2, 3, 4 };
    Box box3; // C2512: no appropriate default constructor available
}

```

Se uma classe não tiver um construtor padrão, uma matriz de objetos dessa classe não poderá ser construída usando apenas a sintaxe de colchete. Por exemplo, dado o bloco de códigos anterior, uma matriz de caixas não pode ser declarada assim:

```
Box boxes[3]; // C2512: no appropriate default constructor available
```

No entanto, você pode usar um conjunto de listas de inicializadores para inicializar uma matriz de objetos Box:

```
Box boxes[3]{ { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };
```

Para obter mais informações, consulte [inicializadores](#).

Copiar construtores

Um *Construtor de cópia* Inicializa um objeto copiando os valores de membro de um objeto do mesmo tipo. Se os membros de sua classe forem todos tipos simples, como valores escalares, o construtor de cópia gerado pelo compilador será suficiente e você não precisará definir o seu próprio. Se sua classe exigir inicialização mais complexa, você precisará implementar um construtor de cópia personalizado. Por exemplo, se um membro de classe for um ponteiro, você precisará definir um construtor de cópia para alocar uma nova memória e copiar os valores do objeto apontado de outro. O construtor de cópia gerado pelo compilador simplesmente copia o ponteiro, para que o novo ponteiro ainda aponte para o local da memória do outro.

Um construtor de cópia pode ter uma destas assinaturas:

```

Box(Box& other); // Avoid if possible--allows modification of other.
Box(const Box& other);
Box(volatile Box& other);
Box(volatile const Box& other);

// Additional parameters OK if they have default values
Box(Box& other, int i = 42, string label = "Box");

```

Ao definir um construtor de cópia, você também deve definir um operador de atribuição de cópia (=). Para obter mais informações, consulte construtores de [atribuição](#) e de [cópia e operadores de atribuição de cópia](#).

Você pode impedir que o objeto seja copiado definindo o construtor de cópia como excluído:

```
Box (const Box& other) = delete;
```

A tentativa de copiar o objeto produz o erro *C2280: tentando fazer referência a uma função excluída*.

Mover construtores

Um *Construtor move* é uma função de membro especial que move a propriedade dos dados de um objeto existente para uma nova variável sem copiar os dados originais. Ele usa uma referência rvalue como seu primeiro parâmetro e os parâmetros adicionais devem ter valores padrão. Os construtores de movimentação podem aumentar significativamente a eficiência do seu programa ao passar objetos grandes.

```
Box(Box&& other);
```

O compilador escolhe um Construtor move em determinadas situações em que o objeto está sendo inicializado por outro objeto do mesmo tipo que está prestes a ser destruído e não precisa mais de seus recursos. O exemplo a seguir mostra um caso quando um Construtor move é selecionado pela resolução de sobrecarga. No construtor que chama `get_Box()`, o valor retornado é um *xValue* (valor de expiração). Ele não está atribuído a nenhuma variável e, portanto, está prestes a sair do escopo. Para fornecer motivação para este exemplo, vamos dar ao box um grande vetor de cadeias de caracteres que representam seu conteúdo. Em vez de copiar o vetor e suas cadeias de caracteres, o Construtor move "rouba" do valor de expiração "box" para que o vetor agora pertença ao novo objeto. A chamada para `std::move` é tudo o que é necessário porque ambas as `vector` e `string` classes e implementam seus próprios construtores de movimentação.

```

#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
using namespace std;

class Box {
public:
    Box() { std::cout << "default" << std::endl; }
    Box(int width, int height, int length)
        : m_width(width), m_height(height), m_length(length)
    {
        std::cout << "int,int,int" << std::endl;
    }
    Box(Box& other)
        : m_width(other.m_width), m_height(other.m_height), m_length(other.m_length)
    {
        std::cout << "copy" << std::endl;
    }
    Box(Box&& other) : m_width(other.m_width), m_height(other.m_height), m_length(other.m_length)
    {
        m_contents = std::move(other.m_contents);
        std::cout << "move" << std::endl;
    }
    int Volume() { return m_width * m_height * m_length; }
    void Add_Item(string item) { m_contents.push_back(item); }
    void Print_Contents()
    {
        for (const auto& item : m_contents)
        {
            cout << item << " ";
        }
    }
private:
    int m_width{ 0 };
    int m_height{ 0 };
    int m_length{ 0 };
    vector<string> m_contents;
};

Box get_Box()
{
    Box b(5, 10, 18); // "int,int,int"
    b.Add_Item("Toupee");
    b.Add_Item("Megaphone");
    b.Add_Item("Suit");

    return b;
}

int main()
{
    Box b; // "default"
    Box b1(b); // "copy"
    Box b2(get_Box()); // "move"
    cout << "b2 contents: ";
    b2.Print_Contents(); // Prove that we have all the values

    char ch;
    cin >> ch; // keep window open
    return 0;
}

```

Se uma classe não definir um construtor de movimentação, o compilador gerará um implícito se não houver nenhum construtor de cópia declarado pelo usuário, operador de atribuição de cópia, operador de atribuição de movimento ou destruidor. Se nenhum construtor de movimentação explícito ou implícito for definido, as

operações que, de outra forma, usarão um construtor de movimentação usarão o construtor de cópia em vez disso. Se uma classe declarar um Construtor mover ou mover o operador de atribuição, o construtor de cópia implicitamente declarado será definido como excluído.

Um construtor de movimentação implicitamente declarado é definido como excluído se qualquer membro que é de tipos de classe não tem um destruidor ou o compilador não pode determinar qual Construtor usar para a operação de movimentação.

Para obter mais informações sobre como escrever um construtor de movimentação não trivial, consulte [mover construtores e mover operadores de atribuição \(C++\)](#).

Construtores padronizados e excluídos explicitamente

Você pode explicitamente *padronizar* construtores de cópia, construtores padrão, construtores de movimentação, operadores de atribuição de cópia, operadores de atribuição de movimento e destruidores. Você pode *excluir* explicitamente todas as funções de membro especiais.

```
class Box
{
public:
    Box2() = delete;
    Box2(const Box2& other) = default;
    Box2& operator=(const Box2& other) = default;
    Box2(Box2&& other) = default;
    Box2& operator=(Box2&& other) = default;
    //...
};
```

Para obter mais informações, consulte [funções explicitamente padronizadas e excluídas](#).

construtores `constexpr`

Um construtor pode ser declarado como `constexpr` se

- Ele é declarado como padrão ou atende a todas as condições para [funções `constexpr`](#) em geral;
- a classe não tem classes base virtuais;
- cada um dos parâmetros é um [tipo literal](#);
- o corpo não é uma função de bloco `try`;
- todos os membros de dados não estáticos e subobjetos de classe base são inicializados;
- se a classe for (a) uma União com membros de variante ou (b) tiver uniões anônimas, somente um dos membros da União será inicializado;
- cada membro de dados não estático do tipo de classe e todos os subobjetos de classe base têm um Construtor `constexpr`

Construtores da lista de inicializadores

Se um construtor usar um `std::initializer_list <T>` como seu parâmetro e quaisquer outros parâmetros tiverem argumentos padrão, esse construtor será selecionado na resolução de sobrecarga quando a classe for instanciada por meio da inicialização direta. Você pode usar o `initializer_list` para inicializar qualquer membro que possa aceitá-lo. Por exemplo, suponha que a classe `Box` (mostrada anteriormente) tenha um `std::vector<string>` membro `m_contents`. Você pode fornecer um construtor como este:

```
Box(initializer_list<string> list, int w = 0, int h = 0, int l = 0)
    : m_contents(list), m_width(w), m_height(h), m_length(l)
{}
```

E, em seguida, criar objetos de caixa como este:

```
Box b{ "apples", "oranges", "pears" }; // or ...
Box b2(initializer_list<string> { "bread", "cheese", "wine" }, 2, 4, 6);
```

Construtores explícitos

Se uma classe tiver um construtor com um único parâmetro ou se todos os parâmetros, exceto um, tiverem um valor padrão, o tipo de parâmetro poderá ser convertido implicitamente no tipo de classe. Por exemplo, se a `Box` classe tiver um construtor como este:

```
Box(int size): m_width(size), m_length(size), m_height(size){}
```

É possível inicializar uma caixa como esta:

```
Box b = 42;
```

Ou passe um int para uma função que usa uma caixa:

```
class ShippingOrder
{
public:
    ShippingOrder(Box b, double postage) : m_box(b), m_postage(postage) {}

private:
    Box m_box;
    double m_postage;
}
//elsewhere...
ShippingOrder so(42, 10.8);
```

Essas conversões podem ser úteis em alguns casos, mas com mais frequência elas podem levar a erros sutis e graves em seu código. Como regra geral, você deve usar a `explicit` palavra-chave em um Construtor (e operadores definidos pelo usuário) para evitar esse tipo de conversão de tipo implícito:

```
explicit Box(int size): m_width(size), m_length(size), m_height(size){}
```

Quando o construtor é explícito, essa linha causa um erro do compilador: `ShippingOrder so(42, 10.8);`. Para obter mais informações, consulte [conversões de tipo definidas pelo usuário](#).

Ordem de construção

Um construtor executa seu trabalho nesta ordem:

1. Chama a classe base e os construtores membros na ordem da declaração.
2. Se a classe for derivada de classes base virtuais, ela inicializará os ponteiros de base virtuais do objeto.
3. Se a classe tiver ou herdar funções virtuais, ela inicializará os ponteiros de função virtual do objeto. Os

ponteiros de função virtual apontam para a tabela de função virtual da classe para permitir a associação de chamadas de função virtual ao código.

4. Executa qualquer código no corpo de sua função.

O exemplo a seguir mostra a ordem em que a classe base e os construtores membros são chamados no construtor para uma classe derivada. Primeiro, o construtor de base é chamado, depois os membros da classe base são inicializados na ordem em que aparecem na declaração de classe e, em seguida, o construtor derivado é chamado.

```
#include <iostream>

using namespace std;

class Contained1 {
public:
    Contained1() { cout << "Contained1 ctor\n"; }
};

class Contained2 {
public:
    Contained2() { cout << "Contained2 ctor\n"; }
};

class Contained3 {
public:
    Contained3() { cout << "Contained3 ctor\n"; }
};

class BaseContainer {
public:
    BaseContainer() { cout << "BaseContainer ctor\n"; }
private:
    Contained1 c1;
    Contained2 c2;
};

class DerivedContainer : public BaseContainer {
public:
    DerivedContainer() : BaseContainer() { cout << "DerivedContainer ctor\n"; }
private:
    Contained3 c3;
};

int main() {
    DerivedContainer dc;
}
```

Esta é a saída:

```
Contained1 ctor
Contained2 ctor
BaseContainer ctor
Contained3 ctor
DerivedContainer ctor
```

Um construtor de classe derivada sempre chama um construtor de classe base, de modo que possa confiar em classes base completamente construídas antes que qualquer trabalho adicional seja feito. Os construtores de classe base são chamados em ordem de derivação — por exemplo, se `ClassA` for derivado de `ClassB`, que é derivado de `ClassC`, o `ClassC` Construtor será chamado primeiro, então o `ClassB` Construtor, em seguida, `ClassA` o construtor.

Se uma classe base não tiver um construtor padrão, você deverá fornecer os parâmetros do construtor de classe base no construtor de classe derivada:

```
class Box {
public:
    Box(int width, int length, int height){
        m_width = width;
        m_length = length;
        m_height = height;
    }

private:
    int m_width;
    int m_length;
    int m_height;
};

class StorageBox : public Box {
public:
    StorageBox(int width, int length, int height, const string label&) : Box(width, length, height){
        m_label = label;
    }

private:
    string m_label;
};

int main(){
    const string aLabel = "aLabel";
    StorageBox sb(1, 2, 3, aLabel);
}
```

Se um construtor gerar uma exceção, a ordem de destruição será a inversa da ordem de construção:

1. O código no corpo da função de construtor é liberado.
2. Os objetos de classe base e de membros são destruídos, na ordem inversa da declaração.
3. Se o construtor não for representante, todos os objetos da classe base e membros completamente construídos serão destruídos. No entanto, como o próprio objeto não está totalmente construído, o destruidor não é executado.

Construtores derivados e inicialização de agregação estendida

Se o construtor de uma classe base for não público, mas estiver acessível a uma classe derivada, em seguida, em **/std: modo c++ 17** no Visual Studio 2017 e posterior, você não poderá usar chaves vazias para inicializar um objeto do tipo derivado.

O exemplo a seguir mostra o comportamento de conformidade do C++14:

```

struct Derived;

struct Base {
    friend struct Derived;
private:
    Base() {}
};

struct Derived : Base {};

Derived d1; // OK. No aggregate init involved.
Derived d2 {}; // OK in C++14: Calls Derived::Derived()
               // which can call Base ctor.

```

No C++ 17, `Derived` agora é considerado um tipo de agregação. Isso significa que a inicialização de `Base` por meio do construtor padrão privado acontece diretamente como parte da regra de inicialização de agregação estendida. Anteriormente, o construtor privado `Base` era chamado por meio do construtor `Derived`, e isso era bem-sucedido devido à declaração `friend`.

O exemplo a seguir mostra o comportamento do C++ 17 no Visual Studio 2017 e posterior no modo `/std: c++17`:

```

struct Derived;

struct Base {
    friend struct Derived;
private:
    Base() {}
};

struct Derived : Base {
    Derived() {} // add user-defined constructor
                 // to call with {} initialization
};

Derived d1; // OK. No aggregate init involved.

Derived d2 {}; // error C2248: 'Base::Base': cannot access
               // private member declared in class 'Base'

```

Construtores para classes que têm várias heranças

Se uma classe for derivada de várias classes base, os construtores de classe base serão chamados na ordem em que estão listados na declaração da classe derivada:

```

#include <iostream>
using namespace std;

class BaseClass1 {
public:
    BaseClass1() { cout << "BaseClass1 ctor\n"; }
};

class BaseClass2 {
public:
    BaseClass2() { cout << "BaseClass2 ctor\n"; }
};

class BaseClass3 {
public:
    BaseClass3() { cout << "BaseClass3 ctor\n"; }
};

class DerivedClass : public BaseClass1,
                     public BaseClass2,
                     public BaseClass3
{
public:
    DerivedClass() { cout << "DerivedClass ctor\n"; }
};

int main() {
    DerivedClass dc;
}

```

O seguinte resultado é esperado:

```

BaseClass1 ctor
BaseClass2 ctor
BaseClass3 ctor
DerivedClass ctor

```

Delegando construtores

Um *Construtor de delegação* chama um Construtor diferente na mesma classe para fazer parte do trabalho de inicialização. Isso é útil quando você tem vários construtores que têm de executar trabalho semelhante. Você pode gravar a lógica principal em um construtor e chamá-la de outras. No exemplo trivial a seguir, Box (int) Delega seu trabalho à caixa (int, int, int):

```

class Box {
public:
    // Default constructor
    Box() {}

    // Initialize a Box with equal dimensions (i.e. a cube)
    Box(int i) : Box(i, i, i); // delegating constructor
    {}

    // Initialize a Box with custom dimensions
    Box(int width, int length, int height)
        : m_width(width), m_length(length), m_height(height)
    {}
    //... rest of class as before
};

```

O objeto criado pelos construtores é inicializado totalmente assim que o construtor é concluído. Para obter mais informações, consulte [delegando construtores](#).

Herdando construtores (C++ 11)

Uma classe derivada pode herdar os construtores de uma classe base direta usando uma `using` declaração, conforme mostrado no exemplo a seguir:

```
#include <iostream>
using namespace std;

class Base
{
public:
    Base() { cout << "Base()" << endl; }
    Base(const Base& other) { cout << "Base(Base&)" << endl; }
    explicit Base(int i) : num(i) { cout << "Base(int)" << endl; }
    explicit Base(char c) : letter(c) { cout << "Base(char)" << endl; }

private:
    int num;
    char letter;
};

class Derived : Base
{
public:
    // Inherit all constructors from Base
    using Base::Base;

private:
    // Can't initialize newMember from Base constructors.
    int newMember{ 0 };
};

int main()
{
    cout << "Derived d1(5) calls: ";
    Derived d1(5);
    cout << "Derived d1('c') calls: ";
    Derived d2('c');
    cout << "Derived d3 = d2 calls: " ;
    Derived d3 = d2;
    cout << "Derived d4 calls: ";
    Derived d4;
}

/* Output:
Derived d1(5) calls: Base(int)
Derived d1('c') calls: Base(char)
Derived d3 = d2 calls: Base(Base&)
Derived d4 calls: Base()*/
```

Visual Studio 2017 e posterior: a `using` instrução em `/std: o modo c++ 17` coloca em escopo todos os construtores da classe base, exceto aqueles que têm uma assinatura idêntica aos construtores na classe derivada. Em geral, é melhor usar construtores de herança quando a classe derivada declara novos membros de dados ou construtores. Consulte também [aprimoramentos no Visual Studio 2017 versão 15,7](#).

Um modelo de classe pode herdar todos os construtores de um argumento de tipo se esse tipo especificar uma classe base:

```
template< typename T >
class Derived : T {
    using T::T;    // declare the constructors from T
    // ...
};
```

Uma classe derivada não pode herdar de várias classes base se essas classes base têm construtores que têm uma assinatura idêntica.

Construtores e classes compostas

As classes que contêm membros de tipo de classe são conhecidas como *classes compostas*. Quando um membro do tipo classe de uma classe composta é criado, o construtor é chamado antes do próprio construtor da classe.

Quando uma classe contida não possuir um construtor padrão, você deverá usar uma lista de inicialização no construtor da classe composta. No exemplo anterior de `StorageBox`, se você alterar o tipo da variável de membro `m_label` para uma nova classe `Label`, deverá chamar o construtor da classe base e inicializar a variável `m_label` no construtor `StorageBox`:

```
class Label {
public:
    Label(const string& name, const string& address) { m_name = name; m_address = address; }
    string m_name;
    string m_address;
};

class StorageBox : public Box {
public:
    StorageBox(int width, int length, int height, Label label)
        : Box(width, length, height), m_label(label){}
private:
    Label m_label;
};

int main(){
// passing a named Label
    Label label1{ "some_name", "some_address" };
    StorageBox sb1(1, 2, 3, label1);

    // passing a temporary label
    StorageBox sb2(3, 4, 5, Label{ "another name", "another address" });

    // passing a temporary label as an initializer list
    StorageBox sb3(1, 2, 3, {"myname", "myaddress"});
}
```

Nesta seção

- [Operadores de construtores de cópia e de atribuição de cópia](#)
- [Operadores de construtores de movimento e de atribuição de movimento](#)
- [Delegação de construtores](#)

Confira também

[Classes e structs](#)

Operadores de construtores de cópia e de atribuição de cópia (C++)

02/09/2020 • 6 minutes to read • [Edit Online](#)

NOTE

A partir do C++ 11, há suporte para dois tipos de atribuição no idioma: *copiar atribuição* e *mover atribuição*. Neste artigo, "atribuição" significa a atribuição de cópia, a menos que explicitamente declarado de outra forma. Para obter informações sobre como mover a atribuição, consulte [mover construtores e mover operadores de atribuição \(C++\)](#).

Tanto a operação de atribuição como a operação de inicialização fazem com que os objetos sejam copiados.

- **Atribuição:** quando o valor de um objeto é atribuído a outro objeto, o primeiro objeto é copiado para o segundo objeto. Portanto,

```
Point a, b;  
...  
a = b;
```

faz com que o valor de `b` seja copiado para `a`.

- **Inicialização:** a inicialização ocorre quando um novo objeto é declarado, quando argumentos são passados para funções por valor ou quando valores são retornados de funções por valor.

Você pode definir a semântica de "cópia" para objetos do tipo classe. Por exemplo, considere este código:

```
TextFile a, b;  
a.Open( "FILE1.DAT" );  
b.Open( "FILE2.DAT" );  
b = a;
```

O código anterior poderia significar "copiar o conteúdo de FILE1.DAT para FILE2.DAT" ou "ignore FILE2.DAT e faça de `b` um segundo manipulador para FILE1.DAT". Você deve anexar a semântica de cópia apropriada a cada classe, da seguinte maneira.

- Usando o operador de atribuição **Operator =** junto com uma referência ao tipo de classe como o tipo de retorno e o parâmetro que é passado por `const` referência — por exemplo
`ClassName& operator=(const ClassName& x);`.
- Usando o construtor de cópia.

Se você não declarar um construtor de cópia, o compilador gerará um construtor de cópia por membro para você. Se você não declarar um operador de atribuição de cópia, o compilador gerará um operador de atribuição de cópia de membro para você. A declaração de um construtor de cópia não suprime o operador de atribuição de cópia gerado pelo compilador, e vice-versa. Se você implementar um deles, recomendamos que você também implemente o outro para que o significado do código seja claro.

O construtor de cópia usa um argumento do tipo *Class-Name &*, em que *Class-Name* é o nome da classe para a qual o construtor é definido. Por exemplo:

```
// spec1_copying_class_objects.cpp
class Window
{
public:
    Window( const Window& ); // Declare copy constructor.
    // ...
};

int main()
{
}
```

NOTE

Crie o tipo de classe de argumento do construtor de cópia `const`, & sempre que possível. Isso evita que o construtor de cópia altere accidentalmente o objeto que está copiando. Ele também permite copiar de `const` objetos.

Construtores de cópia gerados pelo compilador

Os construtores de cópia gerados pelo compilador, como construtores de cópia definidos pelo usuário, têm um único argumento do tipo "Reference to *Class-Name*". Uma exceção é quando todas as classes base e classes de membro têm construtores de cópia declarados como tendo um único argumento do tipo `const Class-Name&`. Nesse caso, o argumento do construtor de cópia gerado pelo compilador também é `const`.

Quando o tipo de argumento para o construtor de cópia não é `const`, a inicialização copiando um `const` objeto gera um erro. O inverso não é verdadeiro: se o argumento for `const`, você poderá inicializar copiando um objeto que não é `const`.

Os operadores de atribuição gerados pelo compilador seguem o mesmo padrão em relação à `const`. Eles usam um único argumento do tipo `Class-Name&`, a menos que os operadores de atribuição em todas as classes base e member tenham argumentos do tipo `const Class-Name&`. Nesse caso, o operador de atribuição gerado pela classe usa um `const` argumento.

NOTE

Quando classes básicas virtuais são inicializadas por construtores de cópia, geradas pelo compilador ou definidas pelo usuário, elas são inicializadas somente uma vez: no ponto em que são construídas.

As implicações são semelhantes às do construtor de cópia. Quando o tipo de argumento não é `const`, a atribuição de um `const` objeto gera um erro. O inverso não é verdadeiro: se um `const` valor for atribuído a um valor que não é `const`, a atribuição terá sucesso.

Para obter mais informações sobre operadores de atribuição sobrecarregados, consulte [atribuição](#).

Operadores de construtores de movimento e de atribuição de movimento (C++)

20/05/2020 • 9 minutes to read • [Edit Online](#)

Este tópico descreve como gravar um *Construtor move* e um operador de atribuição de movimentação para uma classe C++. Um Construtor move permite que os recursos de propriedade de um objeto rvalue sejam movidos para um lvalue sem cópia. Para obter mais informações sobre a semântica de movimentação, consulte [Declarador de referência de rvalue: &&](#).

Este tópico baseia-se na seguinte classe do C++, `MemoryBlock`, que gerencia um buffer de memória.

```
// MemoryBlock.h
#pragma once
#include <iostream>
#include <algorithm>

class MemoryBlock
{
public:

    // Simple constructor that initializes the resource.
    explicit MemoryBlock(size_t length)
        : _length(length)
        , _data(new int[length])
    {
        std::cout << "In MemoryBlock(size_t). length = "
              << _length << "." << std::endl;
    }

    // Destructor.
    ~MemoryBlock()
    {
        std::cout << "In ~MemoryBlock(). length = "
              << _length << ".";

        if (_data != nullptr)
        {
            std::cout << " Deleting resource.";
            // Delete the resource.
            delete[] _data;
        }

        std::cout << std::endl;
    }

    // Copy constructor.
    MemoryBlock(const MemoryBlock& other)
        : _length(other._length)
        , _data(new int[other._length])
    {
        std::cout << "In MemoryBlock(const MemoryBlock&). length = "
              << other._length << ". Copying resource." << std::endl;

        std::copy(other._data, other._data + _length, _data);
    }

    // Copy assignment operator.
    MemoryBlock& operator=(const MemoryBlock& other)
    {
        std::cout << "In operator=(const MemoryBlock&). length = "
              << other._length << ". Copying resource." << std::endl;

        if (_data != nullptr)
            delete[] _data;
        _length = other._length;
        _data = new int[_length];
        std::copy(other._data, other._data + _length, _data);
    }
}
```

```

        << other._length << ". Copying resource." << std::endl;

    if (this != &other)
    {
        // Free the existing resource.
        delete[] _data;

        _length = other._length;
        _data = new int[_length];
        std::copy(other._data, other._data + _length, _data);
    }
    return *this;
}

// Retrieves the length of the data resource.
size_t Length() const
{
    return _length;
}

private:
    size_t _length; // The length of the resource.
    int* _data; // The resource.
};

```

Os procedimentos a seguir descrevem como escrever um construtor de movimentação e um operador de atribuição de movimentação para o exemplo de classe C++.

Para criar um construtor de movimentação para a classe C++

1. Defina um método de construtor vazio que obtenha uma referência rvalue para o tipo de classe como seu parâmetro, como demonstrado no exemplo a seguir:

```

MemoryBlock(MemoryBlock&& other)
    : _data(nullptr)
    , _length(0)
{
}

```

2. No construtor de movimentação, atribua os membros de dados de classe do objeto de origem para o objeto sendo construído:

```

    _data = other._data;
    _length = other._length;

```

3. Atribua os membros de dados do objeto de origem para os valores padrão. Isso impede que o destruidor libere recursos (como a memória) várias vezes:

```

    other._data = nullptr;
    other._length = 0;

```

Para criar um operador de atribuição de movimentação para a classe C++

1. Defina um operador de atribuição vazio que pegue uma referência rvalue para o tipo de classe como seu parâmetro e retorne uma referência para o tipo de classe, como demonstrado no exemplo a seguir:

```

MemoryBlock& operator=(MemoryBlock&& other)
{
}

```

2. No operador de atribuição de movimentação, adicione uma instrução condicional que não execute nenhuma operação se você tentar atribuir o objeto a ele mesmo.

```
if (this != &other)
{
}
```

3. Na instrução condicional, libere quaisquer recursos (como a memória) do objeto ao qual ela está sendo atribuída.

O exemplo a seguir libera o membro `_data` do objeto ao qual está sendo atribuído:

```
// Free the existing resource.
delete[] _data;
```

Siga as etapas 2 e 3 no primeiro procedimento para transferir os membros de dados do objeto de origem para o objeto sendo construído:

```
// Copy the data pointer and its length from the
// source object.
_data = other._data;
_length = other._length;

// Release the data pointer from the source object so that
// the destructor does not free the memory multiple times.
other._data = nullptr;
other._length = 0;
```

4. Retorne uma referência para o objeto atual, conforme mostrado no exemplo a seguir:

```
return *this;
```

Exemplo

O exemplo a seguir mostra o construtor de movimentação completo e o operador de atribuição de movimentação para a classe `MemoryBlock`:

```

// Move constructor.
MemoryBlock(MemoryBlock&& other) noexcept
: _data(nullptr)
, _length(0)
{
    std::cout << "In MemoryBlock(MemoryBlock&&). length = "
        << other._length << ". Moving resource." << std::endl;

    // Copy the data pointer and its length from the
    // source object.
    _data = other._data;
    _length = other._length;

    // Release the data pointer from the source object so that
    // the destructor does not free the memory multiple times.
    other._data = nullptr;
    other._length = 0;
}

// Move assignment operator.
MemoryBlock& operator=(MemoryBlock&& other) noexcept
{
    std::cout << "In operator=(MemoryBlock&&). length = "
        << other._length << "." << std::endl;

    if (this != &other)
    {
        // Free the existing resource.
        delete[] _data;

        // Copy the data pointer and its length from the
        // source object.
        _data = other._data;
        _length = other._length;

        // Release the data pointer from the source object so that
        // the destructor does not free the memory multiple times.
        other._data = nullptr;
        other._length = 0;
    }
    return *this;
}

```

Exemplo

O exemplo a seguir mostra como a semântica de movimentação pode melhorar o desempenho de seus aplicativos. O exemplo adiciona dois elementos a um objeto de vetor e insere um novo elemento entre os dois elementos existentes. A `vector` classe usa a semântica de movimentação para executar a operação de inserção com eficiência movendo os elementos do vetor em vez de copiá-los.

```

// rvalue-references-move-semantics.cpp
// compile with: /EHsc
#include "MemoryBlock.h"
#include <vector>

using namespace std;

int main()
{
    // Create a vector object and add a few elements to it.
    vector<MemoryBlock> v;
    v.push_back(MemoryBlock(25));
    v.push_back(MemoryBlock(75));

    // Insert a new element into the second position of the vector.
    v.insert(v.begin() + 1, MemoryBlock(50));
}

```

Esse exemplo gera a saída a seguir:

```

In MemoryBlock(size_t). length = 25.
In MemoryBlock(MemoryBlock&&). length = 25. Moving resource.
In ~MemoryBlock(). length = 0.
In MemoryBlock(size_t). length = 75.
In MemoryBlock(MemoryBlock&&). length = 75. Moving resource.
In MemoryBlock(MemoryBlock&&). length = 25. Moving resource.
In ~MemoryBlock(). length = 0.
In ~MemoryBlock(). length = 0.
In MemoryBlock(size_t). length = 50.
In MemoryBlock(MemoryBlock&&). length = 50. Moving resource.
In MemoryBlock(MemoryBlock&&). length = 25. Moving resource.
In MemoryBlock(MemoryBlock&&). length = 75. Moving resource.
In ~MemoryBlock(). length = 0.
In ~MemoryBlock(). length = 0.
In ~MemoryBlock(). length = 0.
In ~MemoryBlock(). length = 25. Deleting resource.
In ~MemoryBlock(). length = 50. Deleting resource.
In ~MemoryBlock(). length = 75. Deleting resource.

```

Antes do Visual Studio 2010, este exemplo produziu a seguinte saída:

```

In MemoryBlock(size_t). length = 25.
In MemoryBlock(const MemoryBlock&). length = 25. Copying resource.
In ~MemoryBlock(). length = 25. Deleting resource.
In MemoryBlock(size_t). length = 75.
In MemoryBlock(const MemoryBlock&). length = 25. Copying resource.
In ~MemoryBlock(). length = 25. Deleting resource.
In MemoryBlock(const MemoryBlock&). length = 75. Copying resource.
In ~MemoryBlock(). length = 75. Deleting resource.
In MemoryBlock(size_t). length = 50.
In MemoryBlock(const MemoryBlock&). length = 50. Copying resource.
In MemoryBlock(const MemoryBlock&). length = 50. Copying resource.
In operator=(const MemoryBlock&). length = 75. Copying resource.
In operator=(const MemoryBlock&). length = 50. Copying resource.
In ~MemoryBlock(). length = 50. Deleting resource.
In ~MemoryBlock(). length = 50. Deleting resource.
In ~MemoryBlock(). length = 25. Deleting resource.
In ~MemoryBlock(). length = 50. Deleting resource.
In ~MemoryBlock(). length = 75. Deleting resource.

```

A versão deste exemplo que usa a semântica de movimentação é mais eficiente do que a versão que não usa a semântica de movimentação, pois ela executa menos operações de cópia, alocação de memória e desalocação

de memória.

Programação robusta

Para evitar vazamento de recursos, sempre libere recursos (como a memória, os identificadores de arquivo e os soquetes) no operador de atribuição de movimentação.

Para evitar a destruição irrecuperável de recursos, manipule corretamente a autoatribuição no operador de atribuição de movimentação.

Se você fornecer um construtor de movimentação e um operador de atribuição de movimentação de sua classe, poderá eliminar o código supérfluo escrevendo o construtor de movimentação para chamar o operador de atribuição de movimentação. O exemplo a seguir mostra uma versão revisada do construtor de movimentação que chama o operador de atribuição de movimentação:

```
// Move constructor.
MemoryBlock(MemoryBlock&& other) noexcept
    : _data(nullptr)
    , _length(0)
{
    *this = std::move(other);
}
```

A função `std::move` converte o lvalue `other` em um Rvalue.

Confira também

[Declarador de referência Rvalue: &&](#)
[std::move](#)

Delegação de construtores

15/04/2020 • 3 minutes to read • [Edit Online](#)

Muitas classes têm vários construtores que fazem coisas semelhantes — por exemplo, validam parâmetros:

```
class class_c {
public:
    int max;
    int min;
    int middle;

    class_c() {}
    class_c(int my_max) {
        max = my_max > 0 ? my_max : 10;
    }
    class_c(int my_max, int my_min) {
        max = my_max > 0 ? my_max : 10;
        min = my_min > 0 && my_min < max ? my_min : 1;
    }
    class_c(int my_max, int my_min, int my_middle) {
        max = my_max > 0 ? my_max : 10;
        min = my_min > 0 && my_min < max ? my_min : 1;
        middle = my_middle < max && my_middle > min ? my_middle : 5;
    }
};
```

Você poderia reduzir o código repetitivo adicionando uma função que `class_c` faz toda a validação, mas o código para seria mais fácil de entender e manter se um construtor pudesse delegar parte do trabalho para outro. Para adicionar construtores delegantes, `constructor (. . .) : constructor (. . .)` use a sintaxe:

```
class class_c {
public:
    int max;
    int min;
    int middle;

    class_c(int my_max) {
        max = my_max > 0 ? my_max : 10;
    }
    class_c(int my_max, int my_min) : class_c(my_max) {
        min = my_min > 0 && my_min < max ? my_min : 1;
    }
    class_c(int my_max, int my_min, int my_middle) : class_c (my_max, my_min){
        middle = my_middle < max && my_middle > min ? my_middle : 5;
    }
};
int main() {
    class_c c1{ 1, 3, 2 };
}
```

Ao passar pelo exemplo anterior, observe `class_c(int, int, int)` que o construtor `class_c(int, int)` primeiro chama `class_c(int)` o construtor, que por sua vez chama `.` Cada um dos construtores realiza apenas o trabalho que não é realizado pelos outros construtores.

O primeiro construtor chamado inicializa o objeto para que todos os seus membros sejam inicializados nesse ponto. Você não pode fazer a inicialização de membros em uma construtora que delega para outro construtor,

como mostrado aqui:

```
class class_a {
public:
    class_a() {}
    // member initialization here, no delegate
    class_a(string str) : m_string{ str } {}

    // can't do member initialization here
    // error C3511: a call to a delegating constructor shall be the only member-initializer
    class_a(string str, double dbl) : class_a(str) , m_double{ dbl } {}

    // only member assignment
    class_a(string str, double dbl) : class_a(str) { m_double = dbl; }
    double m_double{ 1.0 };
    string m_string;
};
```

O próximo exemplo mostra o uso de iniciadores não estáticos de membros de dados. Observe que se um construtor também inicializar um determinado membro de dados, o inicializador do membro será substituído:

```
class class_a {
public:
    class_a() {}
    class_a(string str) : m_string{ str } {}
    class_a(string str, double dbl) : class_a(str) { m_double = dbl; }
    double m_double{ 1.0 };
    string m_string{ m_double < 10.0 ? "alpha" : "beta" };
};

int main() {
    class_a a{ "hello", 2.0 }; //expect a.m_double == 2.0, a.m_string == "hello"
    int y = 4;
}
```

A sintaxe da delegação de construtores não impede a criação acidental de recursão de construtores — a Construtora1 chama de Constructor2 — e nenhum erro é lançado até que haja um estouro de pilha. É sua responsabilidade evitar ciclos.

```
class class_f{
public:
    int max;
    int min;

    // don't do this
    class_f() : class_f(6, 3){ }
    class_f(int my_max, int my_min) : class_f() { }
};
```

Destruidores (C++)

02/09/2020 • 13 minutes to read • [Edit Online](#)

Um destruidor é uma função membro que é invocada automaticamente quando o objeto sai do escopo ou destruído explicitamente por uma chamada para `delete`. Um destruidor tem o mesmo nome que a classe, precedida por um til (`~`). Por exemplo, o destruidor da classe `String` é declarado: `~String()`.

Se você não definir um destruidor, o compilador fornecerá um padrão; para muitas classes, isso é suficiente. Você só precisa definir um destruidor personalizado quando a classe armazena identificadores para recursos do sistema que precisam ser liberados ou ponteiros que possuem a memória para a qual apontam.

Observe a considerar a seguinte declaração de uma classe `String`:

```
// spec1_destructors.cpp
#include <string>

class String {
public:
    String( char *ch ); // Declare constructor
    ~String(); // and destructor.
private:
    char *_text;
    size_t sizeOfText;
};

// Define the constructor.
String::String( char *ch ) {
    sizeOfText = strlen( ch ) + 1;

    // Dynamically allocate the correct amount of memory.
    _text = new char[ sizeOfText ];

    // If the allocation succeeds, copy the initialization string.
    if( _text )
        strcpy_s( _text, sizeOfText, ch );
}

// Define the destructor.
String::~String() {
    // Deallocate the memory that was previously reserved
    // for this string.
    delete[] _text;
}

int main() {
    String str("The piper in the glen...");
}
```

No exemplo anterior, o destruidor `String::~String` usa o `delete` operador para desalocar o espaço alocado dinamicamente para armazenamento de texto.

Declarando destruidores

Os destruidores são funções com o mesmo nome da classe, mas precedidos por um til (`~`)

Várias regras controlam a declaração de destruidores. Destruidores:

- Não aceitam argumentos.

- Não retornar um valor (ou `void`).
- Não pode ser declarado como `const` , `volatile` ou `static` . No entanto, eles podem ser invocados para a destruição de objetos declarados como `const` , `volatile` ou `static` .
- Pode ser declarado como `virtual` . Usando destruidores virtuais, você pode destruir objetos sem conhecer o tipo deles — o destruidor correto para o objeto é invocado usando o mecanismo de função virtual. Observe que os destruidores também podem ser declarados como funções virtuais puras para classes abstratas.

Usando destruidores

Os destruidores são chamados quando ocorre um dos seguintes eventos:

- Um objeto local (automático) com escopo de bloco sai do escopo.
- Um objeto alocado usando o `new` operador é explicitamente desalocado usando `delete` .
- O tempo de vida de um objeto temporário termina.
- Um programa é encerrado e existem objetos globais ou estáticos.
- O destruidor é chamado explicitamente usando o nome totalmente qualificado da função de destruidor.

Os destruidores podem chamar funções de membro de classe e acessar dados de membros de classe livremente.

Há duas restrições sobre o uso de destruidores:

- Você não pode obter seu endereço.
- Classes derivadas não herdam o destruidor de sua classe base.

Ordem de destruição

Quando um objeto sai do escopo ou é excluído, a sequência de eventos em sua destruição completa é a seguinte:

1. O destruidor da classe é chamado e o corpo da função de destruidor é executado.
2. Os destruidores para objetos dos membros não estáticos são chamados na ordem inversa em que aparecem na declaração da classe. A lista de inicialização de membro opcional usada na construção desses membros não afeta a ordem de construção ou destruição.
3. Os destruidores para classes de base não virtuais são chamados na ordem inversa da declaração.
4. Os destruidores de classes base virtuais são chamados na ordem inversa da declaração.

```

// order_of_destruction.cpp
#include <cstdio>

struct A1      { virtual ~A1() { printf("A1 dtor\n"); } };
struct A2 : A1 { virtual ~A2() { printf("A2 dtor\n"); } };
struct A3 : A2 { virtual ~A3() { printf("A3 dtor\n"); } };

struct B1      { ~B1() { printf("B1 dtor\n"); } };
struct B2 : B1 { ~B2() { printf("B2 dtor\n"); } };
struct B3 : B2 { ~B3() { printf("B3 dtor\n"); } };

int main() {
    A1 * a = new A3;
    delete a;
    printf("\n");

    B1 * b = new B3;
    delete b;
    printf("\n");

    B3 * b2 = new B3;
    delete b2;
}

Output: A3 dtor
A2 dtor
A1 dtor

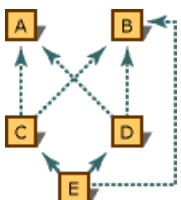
B1 dtor

B3 dtor
B2 dtor
B1 dtor

```

Classes base virtuais

Os destruidores para as classes base virtuais são chamados na ordem inversa de sua aparência em um grafo acíclico direcionado (profundidade-primeiro, da esquerda para a direita, passagem de ordem dupla). A figura a seguir descreve um grafo de herança.



Grafo de herança que mostra as classes base virtuais

A lista a seguir mostra os cabeçalhos de classe das classes mostradas.

```

class A
class B
class C : virtual public A, virtual public B
class D : virtual public A, virtual public B
class E : public C, public D, virtual public B

```

Para determinar a ordem de destruição das classes base virtuais de um objeto do tipo `E`, o compilador cria uma lista aplicando o seguinte algoritmo:

1. Percorrer o gráfico à esquerda, a partir do ponto mais profundo do gráfico (nesse caso, `E`).
2. Executar percurso à esquerda até que todos os nós sejam visitados. Tomar nota do nome do nó atual.

3. Retornar ao nó anterior (para baixo e à direita) para descobrir se o nó que está sendo recordado é uma classe base virtual.
4. Se o nó recordado for uma classe base virtual, verificar a lista para ver se ela já foi inserida. Se não for uma classe base virtual, ignora-la.
5. Se o nó recordado ainda não estiver na lista, adiciona-lo à parte inferior da lista.
6. Percorrer o gráfico acima e ao longo do caminho seguinte à direita.
7. Vá para a etapa 2.
8. Quando o último caminho acima tiver sido esgotado, tomar nota o nome do nó atual.
9. Vá para a etapa 3.
10. Continuar esse processo até que o nó final seja novamente o nó atual.

Portanto, para a classe `E`, a ordem de destruição é:

1. A classe base não virtual `E`.
2. A classe base não virtual `D`.
3. A classe base não virtual `C`.
4. A classe base virtual `B`.
5. A classe base virtual `A`.

Esse processo gera uma lista ordenada de entradas exclusivas. Nenhum nome de classe aparece duas vezes.

Depois que a lista é construída, ela é percorrida na ordem inversa e o destruidor de cada uma das classes na lista, da última até a primeira, é chamado.

A ordem de construção ou de destruição primeiro é muito importante quando os construtores ou os destruidores em uma classe se baseiam em outro componente criado antes ou que dura por mais tempo — por exemplo, se o destruidor de `A` (mostrada na figura anterior) depende que `B` ainda esteja presente quando o código for executado, ou vice-versa.

Essas interdependências entre classes em um grafo de herança são inherentemente perigosas, pois as classes derivadas posteriormente podem alterar qual é o caminho mais à esquerda, alterando assim a ordem de construção e destruição.

Classes base não virtuais

Os destruidores para classes de base não virtuais são chamados na ordem inversa em que os nomes de classe base são declarados. Considere a seguinte declaração de classe:

```
class MultInherit : public Base1, public Base2
...
```

No exemplo anterior, o destruidor de `Base2` é chamado antes de destruidor de `Base1`.

Chamadas explícitas de destruidores

Chamar um destruidor de forma explícita raramente é necessário. No entanto, pode ser útil realizar a limpeza dos objetos colocados em endereços absolutos. Esses objetos são normalmente alocados usando um operador definido pelo usuário `new` que usa um argumento de posicionamento. O `delete` operador não pode desalocar esta memória porque ela não está alocada do armazenamento gratuito (para obter mais informações, consulte [os operadores New e Delete](#)). Entretanto, uma chamada para o destruidor pode realizar uma limpeza apropriada. Para

chamar explicitamente o destruidor de um objeto, `s`, da classe `String`, use uma das seguintes instruções:

```
s.String::~String();      // non-virtual call
ps->String::~String();   // non-virtual call

s.~String();              // Virtual call
ps->~String();           // Virtual call
```

A notação de chamadas explícitas aos destruidores, mostrada anteriormente, pode ser usada independentemente de o tipo definir um destruidor. Isso permite fazer essas chamadas explícitas sem saber se há um destruidor definido para o tipo. Uma chamada explícita para um destruidor que não tem nenhum definido não tem efeito.

Programação robusta

Uma classe precisa de um destruidor se adquirir um recurso e gerenciar o recurso com segurança, provavelmente precisará implementar um construtor de cópia e uma atribuição de cópia.

Se essas funções especiais não forem definidas pelo usuário, elas serão definidas implicitamente pelo compilador. Os construtores e operadores de atribuição gerados implicitamente executam a cópia superficial, `memberwise`, que é quase certamente errado se um objeto estiver gerenciando um recurso.

No próximo exemplo, o construtor de cópia gerado implicitamente fará os ponteiros `str1.text` e `str2.text` se referirem à mesma memória e, quando retornarmos `copy_strings()`, essa memória será excluída duas vezes, o que é um comportamento indefinido:

```
void copy_strings()
{
    String str1("I have a sense of impending disaster...");
    String str2 = str1; // str1.text and str2.text now refer to the same object
} // delete[] _text; deallocates the same memory twice
// undefined behavior
```

Definir explicitamente um destruidor, um construtor de cópia ou um operador de atribuição de cópia impede a definição implícita do construtor de movimentação e o operador de atribuição de movimentação. Nesse caso, a falha ao fornecer operações de movimentação é geralmente, se a cópia for cara, uma oportunidade de otimização perdida.

Confira também

[Copiar construtores e operadores de atribuição de cópia](#)

[Mover construtores e mover operadores de atribuição](#)

Visão geral das funções de membro

02/09/2020 • 3 minutes to read • [Edit Online](#)

As funções membro são estáticas ou não estáticas. O comportamento de funções de membro estáticos difere de outras funções de membro porque as funções de membro static não têm nenhum `this` argumento implícito. As funções de membro não estático têm um `this` ponteiro. As funções membro, sejam elas estáticas ou não estáticas, podem ser definidas dentro ou fora da declaração da classe.

Se uma função membro for definida dentro de uma declaração de classe, ela será tratada como uma função embutida, e não há necessidade de qualificar o nome da função com o nome da sua classe. Embora as funções definidas nas declarações de classe já sejam tratadas como funções embutidas, você pode usar a `inline` palavra-chave para documentar o código.

Este é um exemplo de declaração de uma função dentro de uma declaração de classe:

```
// overview_of_member_functions1.cpp
class Account
{
public:
    // Declare the member function Deposit within the declaration
    // of class Account.
    double Deposit( double HowMuch )
    {
        balance += HowMuch;
        return balance;
    }
private:
    double balance;
};

int main()
{
}
```

Se a definição de uma função de membro estiver fora da declaração de classe, ela será tratada como uma função embutida somente se ela estiver explicitamente declarada como `inline`. Além disso, o nome da função na definição deve ser qualificado com o nome da sua classe usando o operador de resolução de escopo (`::`).

O exemplo a seguir é idêntico à declaração de classe `Account` anterior, com exceção da função `Deposit`, que é definida fora de declaração da classe:

```
// overview_of_member_functions2.cpp
class Account
{
public:
    // Declare the member function Deposit but do not define it.
    double Deposit( double HowMuch );
private:
    double balance;
};

inline double Account::Deposit( double HowMuch )
{
    balance += HowMuch;
    return balance;
}

int main()
{
}
```

NOTE

Embora as funções membro possam ser definidas dentro de uma declaração de classe ou separadamente, nenhuma função membro pode ser adicionada a uma classe depois que a classe é definida.

As classes que contêm funções membro podem ter muitas declarações, mas as próprias funções membro devem ter apenas uma definição em um programa. Várias definições geram uma mensagem de erro em tempo de vinculação. Se uma classe contiver definições de função embutidas, as definições de função devem ser idênticas para observar a regra de "uma definição".

Especificador virtual

25/03/2020 • 2 minutes to read • [Edit Online](#)

A palavra-chave **virtual** só pode ser aplicada a funções de membros de classes não estáticas. Significa que a associação de chamadas à função é adiada até o tempo de execução. Para obter mais informações, consulte [funções virtuais](#).

substituir especificador

25/03/2020 • 2 minutes to read • [Edit Online](#)

Você pode usar a palavra-chave **override** para designar funções de membro que substituem uma função virtual em uma classe base.

Sintaxe

```
function-declaration override;
```

Comentários

a **substituição** é contextual e tem um significado especial apenas quando é usada após uma declaração de função de membro; caso contrário, não é uma palavra-chave reservada.

Exemplo

Use **override** para ajudar a evitar o comportamento de herança acidental em seu código. O exemplo a seguir mostra onde, sem usar **override**, o comportamento da função membro da classe derivada pode não ter sido planejado. O compilador não emite erros para esse código.

```
class BaseClass
{
    virtual void funcA();
    virtual void funcB() const;
    virtual void funcC(int = 0);
    void funcD();
};

class DerivedClass: public BaseClass
{
    virtual void funcA(); // ok, works as intended

    virtual void funcB(); // DerivedClass::funcB() is non-const, so it does not
                         // override BaseClass::funcB() const and it is a new member function

    virtual void funcC(double = 0.0); // DerivedClass::funcC(double) has a different
                                    // parameter type than BaseClass::funcC(int), so
                                    // DerivedClass::funcC(double) is a new member function
};
```

Quando você usa **override**, o compilador gera erros em vez de criar silenciosamente novas funções de membro.

```
class BaseClass
{
    virtual void funcA();
    virtual void funcB() const;
    virtual void funcC(int = 0);
    void funcD();
};

class DerivedClass: public BaseClass
{
    virtual void funcA() override; // ok

    virtual void funcB() override; // compiler error: DerivedClass::funcB() does not
                                  // override BaseClass::funcB() const

    virtual void funcC( double = 0.0 ) override; // compiler error:
                                                // DerivedClass::funcC(double) does not
                                                // override BaseClass::funcC(int)

    void funcD() override; // compiler error: DerivedClass::funcD() does not
                          // override the non-virtual BaseClass::funcD()
};


```

Para especificar que as funções não podem ser substituídas e que as classes não podem ser herdadas, use a palavra-chave **final** .

Confira também

[Especificador final](#)

[Palavras-chave](#)

especificador final

25/03/2020 • 2 minutes to read • [Edit Online](#)

Você pode usar a palavra-chave **final** para designar funções virtuais que não podem ser substituídas em uma classe derivada. Também é possível usá-la para designar classes que não podem ser herdadas.

Sintaxe

```
function-declaration final;  
class class-name final base-classes
```

Comentários

final é contextual e tem um significado especial apenas quando é usado após uma declaração de função ou nome de classe; caso contrário, não é uma palavra-chave reservada.

Quando **final** é usado em declarações de classe, `base-classes` é uma parte opcional da declaração.

Exemplo

O exemplo a seguir usa a palavra-chave **final** para especificar que uma função virtual não pode ser substituída.

```
class BaseClass  
{  
    virtual void func() final;  
};  
  
class DerivedClass: public BaseClass  
{  
    virtual void func(); // compiler error: attempting to  
                        // override a final function  
};
```

Para obter informações sobre como especificar que as funções de membro podem ser substituídas, consulte [especificador de substituição](#).

O exemplo a seguir usa a palavra-chave **final** para especificar que uma classe não pode ser herdada.

```
class BaseClass final  
{  
};  
  
class DerivedClass: public BaseClass // compiler error: BaseClass is  
                                    // marked as non-inheritable  
{  
};
```

Confira também

[Palavras-chave](#)

[Especificador override](#)

Herança (C++)

02/09/2020 • 2 minutes to read • [Edit Online](#)

Esta seção explica como usar classes derivadas para gerar programas extensíveis.

Visão geral

Novas classes podem ser derivadas de classes existentes usando um mecanismo chamado "herança" (consulte as informações que começam em [herança única](#)). As classes que são usadas para derivação são chamadas "classes base" de uma classe derivada específica. Uma classe derivada é declarada mediante a seguinte sintaxe:

```
class Derived : [virtual] [access-specifier] Base
{
    // member list
};
class Derived : [virtual] [access-specifier] Base1,
    [virtual] [access-specifier] Base2, . . .
{
    // member list
};
```

Depois da marca (nome) da classe, aparecem dois-pontos seguidos por uma lista de especificações de base. As classes base nomeadas dessa forma devem ter sido declaradas anteriormente. As especificações básicas podem conter um especificador de acesso, que é uma das palavras- `public` chave `protected` ou `private` . Esses especificadores de acesso aparecem antes do nome da classe base e só se aplicam a essa classe base. Esses especificadores controlam a permissão da classe derivada para usar os membros da classe base. Consulte [controle de acesso a membro](#) para obter informações sobre o acesso a membros de classe base. Se o especificador de acesso for omitido, o acesso a essa base será considerado `private` . As especificações básicas podem conter a palavra-chave `virtual` para indicar a herança virtual. Essa palavra-chave pode aparecer antes ou depois do especificador de acesso, se houver. Se for usada a herança virtual, a classe base será conhecida como uma classe base virtual.

É possível especificar várias classes base, separadas por vírgulas. Se uma única classe base for especificada, o modelo de herança será [herança única](#). Se mais de uma classe base for especificada, o modelo de herança será chamado de [várias heranças](#).

Os tópicos a seguir estão incluídos:

- [Herança única](#)
- [Várias classes base](#)
- [Funções virtuais](#)
- [Substituições explícitas](#)
- [Classes abstratas](#)
- [Resumo das regras de escopo](#)

As palavras-chave `__super` e `__interface` estão documentadas nesta seção.

Confira também

Funções virtuais

02/09/2020 • 6 minutes to read • [Edit Online](#)

Uma função virtual é uma função membro que espera-se que seja redefinida em classes derivadas. Quando você faz referência um objeto da classe derivada usando um ponteiro ou uma referência à classe base, pode chamar uma função virtual para esse objeto e executar a versão da classe derivada da função.

As funções virtuais asseguram que a função correta seja chamada para um objeto, independentemente da expressão usada para fazer a chamada de função.

Suponha que uma classe base contenha uma função declarada como `virtual` e uma classe derivada defina a mesma função. A função a partir da classe derivada é chamada para objetos da classe derivada, mesmo se ela for chamada usando um ponteiro ou uma referência à classe base. O exemplo a seguir mostra uma classe base que fornece uma implementação da função de `PrintBalance` e de duas classes derivadas

```
// deriv_VirtualFunctions.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

class Account {
public:
    Account( double d ) { _balance = d; }
    virtual ~Account() {}
    virtual double GetBalance() { return _balance; }
    virtual void PrintBalance() { cerr << "Error. Balance not available for base type." << endl; }
private:
    double _balance;
};

class CheckingAccount : public Account {
public:
    CheckingAccount(double d) : Account(d) {}
    void PrintBalance() { cout << "Checking account balance: " << GetBalance() << endl; }
};

class SavingsAccount : public Account {
public:
    SavingsAccount(double d) : Account(d) {}
    void PrintBalance() { cout << "Savings account balance: " << GetBalance(); }
};

int main() {
    // Create objects of type CheckingAccount and SavingsAccount.
    CheckingAccount checking( 100.00 );
    SavingsAccount savings( 1000.00 );

    // Call PrintBalance using a pointer to Account.
    Account *pAccount = &checking;
    pAccount->PrintBalance();

    // Call PrintBalance using a pointer to Account.
    pAccount = &savings;
    pAccount->PrintBalance();
}
```

No código anterior, as chamadas para `PrintBalance` são idênticas, com exceção do objeto ao qual `pAccount` aponta. Como `PrintBalance` é `virtual`, a versão da função definida para cada objeto é chamada. A função

`PrintBalance` nas classes derivadas `CheckingAccount` e `SavingsAccount` "substitui" a função na classe base `Account`.

Se uma classe que não fornece uma implementação substituta da função `PrintBalance` for chamada, a implementação padrão da classe base `Account` será usada.

As funções nas classes derivadas substituem as funções virtuais nas classes base apenas se o tipo for o mesmo. Uma função em uma classe derivada não pode ser diferente de uma função virtual em uma classe base apenas no tipo de retorno; a lista de argumentos também deve ser diferente.

Ao chamar uma função usando ponteiros ou referências, as seguintes regras se aplicam:

- Uma chamada a uma função virtual é resolvida de acordo com o tipo subjacente de objeto para o qual ela é chamada.
- Uma chamada a uma função não virtual é resolvida de acordo com o tipo de ponteiro ou de referência.

O exemplo a seguir mostra como as funções virtuais e não virtuais se comportam quando chamadas por ponteiros:

```

// deriv_VirtualFunctions2.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

class Base {
public:
    virtual void NameOf(); // Virtual function.
    void InvokingClass(); // Nonvirtual function.
};

// Implement the two functions.
void Base::NameOf() {
    cout << "Base::NameOf\n";
}

void Base::InvokingClass() {
    cout << "Invoked by Base\n";
}

class Derived : public Base {
public:
    void NameOf(); // Virtual function.
    void InvokingClass(); // Nonvirtual function.
};

// Implement the two functions.
void Derived::NameOf() {
    cout << "Derived::NameOf\n";
}

void Derived::InvokingClass() {
    cout << "Invoked by Derived\n";
}

int main() {
    // Declare an object of type Derived.
    Derived aDerived;

    // Declare two pointers, one of type Derived * and the other
    // of type Base *, and initialize them to point to aDerived.
    Derived *pDerived = &aDerived;
    Base    *pBase   = &aDerived;

    // Call the functions.
    pBase->NameOf();           // Call virtual function.
    pBase->InvokingClass();    // Call nonvirtual function.
    pDerived->NameOf();        // Call virtual function.
    pDerived->InvokingClass(); // Call nonvirtual function.
}

```

```

Derived::NameOf
Invoked by Base
Derived::NameOf
Invoked by Derived

```

Observe que, independentemente da função `NameOf` ser chamada por um ponteiro para `Base` ou um ponteiro para `Derived`, ela chama a função para `Derived`. Ela chama a função para `Derived` porque `NameOf` é uma função virtual, e `pBase` e `pDerived` apontam para um objeto do tipo `Derived`.

Como as funções virtuais são chamadas apenas para objetos de tipos de classe, você não pode declarar funções globais ou estáticas como `virtual`.

A `virtual` palavra-chave pode ser usada ao declarar funções de substituição em uma classe derivada, mas é desnecessária; as substituições de funções virtuais são sempre virtuais.

As funções virtuais em uma classe base devem ser definidas, a menos que sejam declaradas usando o *especificador puro*. (Para obter mais informações sobre funções virtuais puras, consulte [classes abstratas](#).)

O mecanismo de chamada de funções virtuais pode ser suprimido ao qualificar explicitamente o nome da função usando o operador scope-resolution (`::`). Veja o exemplo anterior que envolve a classe `Account`. Para chamar `PrintBalance` na classe base, use um código como este:

```
CheckingAccount *pChecking = new CheckingAccount( 100.00 );  
  
pChecking->Account::PrintBalance(); // Explicit qualification.  
  
Account *pAccount = pChecking; // Call Account::PrintBalance  
  
pAccount->Account::PrintBalance(); // Explicit qualification.
```

As duas chamadas a `PrintBalance` no exemplo anterior suprimem o mecanismo de chamada de função virtual.

Herança única

02/09/2020 • 7 minutes to read • [Edit Online](#)

Na "herança única", uma forma comum de herança, as classes têm apenas uma classe base. Considere a relação ilustrada na figura a seguir.

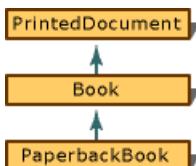


Gráfico simples de herança única

Observe a progressão de geral a específico na figura. Outro atributo comum encontrado no design da maioria das hierarquias de classes é que a classe derivada tem um "tipo de" relação com a classe base. Na figura, `Book` é um tipo de `PrintedDocument`, e `PaperbackBook` é um tipo de `book`.

Outro item de observação na figura: `Book` é uma classe derivada (de `PrintedDocument`) e uma classe base (`PaperbackBook` é derivado de `Book`). Uma declaração estrutural de uma hierarquia de classes é mostrada no exemplo a seguir:

```
// deriv_SingleInheritance.cpp
// compile with: /LD
class PrintedDocument {};

// Book is derived from PrintedDocument.
class Book : public PrintedDocument {};

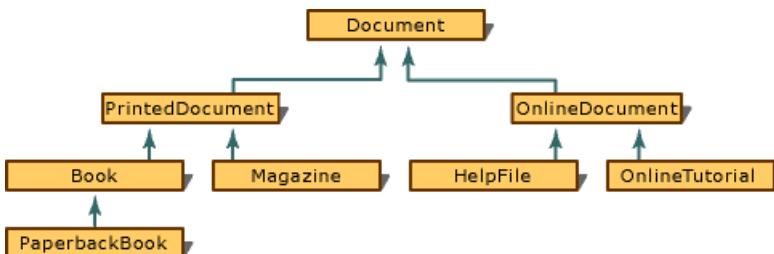
// PaperbackBook is derived from Book.
class PaperbackBook : public Book {};
```

`PrintedDocument` é considerado uma classe "base direta" de `Book`; é uma classe "base indireta" de `PaperbackBook`. A diferença é que uma classe base direta aparece na lista base de uma declaração de classe, e uma base indireta, não.

A classe base da qual cada classe é derivada é declarada antes da declaração da classe derivada. Não é suficiente para fornecer uma declaração de referência de encaminhamento para uma classe base; deve ser uma declaração completa.

No exemplo anterior, o especificador de acesso `public` é usado. O significado de herança pública, protegida e privada é descrito em [controle de acesso de membro](#).

Uma classe pode servir como a classe base para muitas aulas específicas, como ilustrado na figura a seguir.



Exemplo de gráfico acíclico direcionado

No diagrama mostrado acima, chamado "grafo direcionado acíclico" (ou "DAG"), algumas das classes são classes

base para mais de uma classe derivada. No entanto, o contrário não é verdadeiro: há apenas uma classe base direta para qualquer classe derivada especificada. O gráfico na figura denota uma estrutura de "herança única".

NOTE

Os grafos direcionados acíclicos não são exclusivos da herança única. Também são usados para denotar gráficos de herança múltipla.

Na herança, a classe derivada contém os membros da classe base mais os novos membros que você adicionar. Como resultado, uma classe derivada pode se referir a membros da classe base (a menos que esses membros sejam redefinidos na classe derivada). O operador de resolução de escopo (`::`) pode ser usado para fazer referência a membros de classes base diretas e indiretas quando esses membros são redefinidos na classe derivada. Considere este exemplo:

```
// deriv_SingleInheritance2.cpp
// compile with: /EHsc /c
#include <iostream>
using namespace std;
class Document {
public:
    char *Name;    // Document name.
    void PrintNameOf(); // Print name.
};

// Implementation of PrintNameOf function from class Document.
void Document::PrintNameOf() {
    cout << Name << endl;
}

class Book : public Document {
public:
    Book( char *name, long pagecount );
private:
    long PageCount;
};

// Constructor from class Book.
Book::Book( char *name, long pagecount ) {
    Name = new char[ strlen( name ) + 1 ];
    strcpy_s( Name, strlen( Name ), name );
    PageCount = pagecount;
}
```

Observe que o construtor para `Book` (`Book::Book`) tem acesso ao membro de dados, `Name`. Em um programa, um objeto do tipo `Book` pode ser criado e usado como segue:

```
// Create a new object of type Book. This invokes the
// constructor Book::Book.
Book LibraryBook( "Programming Windows, 2nd Ed", 944 );

...
// Use PrintNameOf function inherited from class Document.
LibraryBook.PrintNameOf();
```

Como o exemplo acima mostra, o membro da classe e os dados e as funções herdados são usados de forma idêntica. Se a implementação para chamadas da classe `Book` para uma reimplementação da função `PrintNameOf`, a função que pertence à classe `Document` pode ser chamada somente usando o operador de resolução de escopo (`::`).

```

// deriv_SingleInheritance3.cpp
// compile with: /EHsc /LD
#include <iostream>
using namespace std;

class Document {
public:
    char *Name;           // Document name.
    void PrintNameOf() {} // Print name.
};

class Book : public Document {
    Book( char *name, long pagecount );
    void PrintNameOf();
    long PageCount;
};

void Book::PrintNameOf() {
    cout << "Name of book: ";
    Document::PrintNameOf();
}

```

Os ponteiros e as referências a classes derivadas podem ser convertidos implicitamente em ponteiros e as referências às suas classes base, se houver uma classe base acessível inequívoca. O código a seguir demonstra esse conceito usando ponteiros (o mesmo princípio se aplica às referências):

```

// deriv_SingleInheritance4.cpp
// compile with: /W3
struct Document {
    char *Name;
    void PrintNameOf() {}
};

class PaperbackBook : public Document {};

int main() {
    Document * DocLib[10]; // Library of ten documents.
    for (int i = 0 ; i < 5 ; i++)
        DocLib[i] = new Document;
    for (int i = 5 ; i < 10 ; i++)
        DocLib[i] = new PaperbackBook;
}

```

No exemplo anterior, diferentes tipos são criados. No entanto, como todos esses tipos são derivados da classe `Document`, há uma conversão implícita em `Document *`. Como resultado, `DocLib` é uma "lista heterogênea" (uma lista em que nem todos os objetos são do mesmo tipo) que contém tipos diferentes de objetos.

Como a classe `Document` tem uma função `PrintNameOf`, ela pode imprimir o nome de cada livro na biblioteca, embora possa omitir informações específicas do tipo de documento (número de páginas de `Book`, número de bytes de `HelpFile`, e assim por diante).

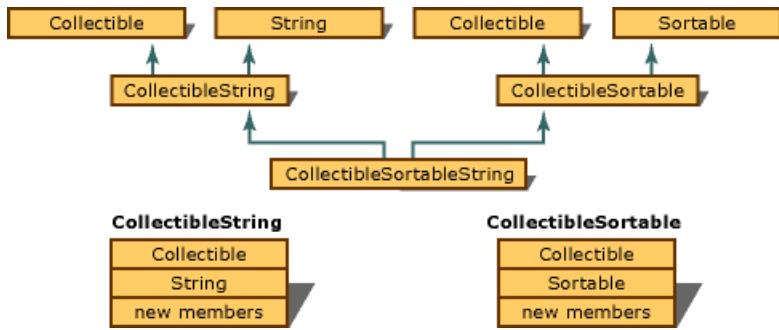
NOTE

Forçar a classe base a implementar uma função como `PrintNameOf` normalmente não é o melhor design. As [funções virtuais](#) oferecem outras alternativas de design.

Classes base

25/03/2020 • 2 minutes to read • [Edit Online](#)

O processo de herança cria uma nova classe derivada que é composta dos membros da classe base (ou das classes base), mais os novos membros adicionados pela classe derivada. Em uma herança múltipla, é possível construir um gráfico de herança em que a mesma classe base faz parte de mais de uma das classes derivadas. A figura a seguir mostra um gráfico desse tipo.



Várias instâncias de uma única classe base

Na figura, são mostradas representações pictóricas dos componentes de `CollectibleString` e `CollectibleSortable`. No entanto, a classe base, `Collectible`, está em `CollectibleSortableString` pelo caminho de `CollectibleString` e pelo caminho de `CollectibleSortable`. Para eliminar essa redundância, essas classes podem ser declaradas como classes base virtuais quando são herdadas.

Várias classes base

02/09/2020 • 15 minutes to read • [Edit Online](#)

Uma classe pode ser derivada de mais de uma classe base. Em um modelo de várias heranças (em que classes são derivadas de mais de uma classe base), as classes base são especificadas usando o elemento de gramática de *lista de base*. Por exemplo, a declaração de classe de `CollectionOfBook`, derivada de `Collection` e de `Book`, pode ser especificada:

```
// deriv_MultipleBaseClasses.cpp
// compile with: /LD
class Collection {
};
class Book {};
class CollectionOfBook : public Book, public Collection {
    // New members
};
```

A ordem na qual as classes base são especificadas não é significativa, exceto em determinados casos nos quais os construtores e os destruidores são invocados. Nesses casos, a ordem na qual as classes base são especificadas afeta o seguinte:

- A ordem na qual ocorre a inicialização pelo construtor. Se seu código depende da parte `Book` de `CollectionOfBook` para ser inicializado antes da parte `Collection`, a ordem de especificação é significante. A inicialização ocorre na ordem em que as classes são especificadas na *lista de base*.
- A ordem na qual os destruidores são chamados para limpeza. Novamente, se for necessário que uma "parte" específica da classe esteja presente quando outra parte for destruída, a ordem é importante. Os destruidores são chamados na ordem inversa das classes especificadas na *lista de base*.

NOTE

A ordem de especificação das classes base pode afetar o layout de memória da classe. Não tome decisões de programação com base na ordem dos membros base na memória.

Ao especificar a *lista de base*, você não pode especificar o mesmo nome de classe mais de uma vez. No entanto, uma classe pode ser uma base indireta para uma classe derivada mais de uma vez.

Classes base virtuais

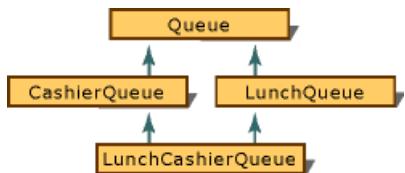
Porque uma classe pode ser uma classe base indireta a uma classe derivada mais de uma vez, C++ fornece uma maneira de otimizar a forma como funcionam as classes base. As classes base virtuais oferecem uma maneira de economizar espaço e evitar ambiguidades nas hierarquias de classes que usam a herança múltipla.

Cada objeto não virtual contém uma cópia dos membros de dados definidos na classe base. Essa duplicação perde espaço e exige que você especifique que cópia dos membros da classe base você quer sempre que os acessa.

Quando uma classe base é especificada como base virtual, ela pode atuar como uma base indireta mais de uma vez sem duplicação de seus membros de dados. Uma única cópia dos membros de dados é compartilhada por todas as classes base que ela usa como base virtual.

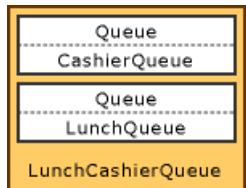
Ao declarar uma classe base virtual, a `virtual` palavra-chave é exibida nas listas base das classes derivadas.

Considere a hierarquia de classes na figura a seguir, que ilustra uma linha simulada de almoço.



Grafo de linha de almoço simulado

Na figura, `Queue` é a classe base para `CashierQueue` e `LunchQueue`. No entanto, quando as duas classes são combinadas para formar `LunchCashierQueue`, o seguinte problema ocorre: a nova classe contém dois subobjetos do tipo `Queue`, um de `CashierQueue` e o outro de `LunchQueue`. A figura a seguir mostra o layout conceitual de memória (o layout real de memória pode ser otimizado).

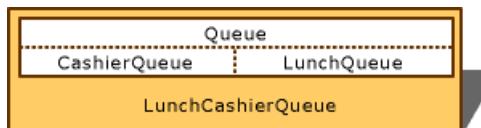


Objeto de linha de almoço simulado

Observe que há dois subobjetos `Queue` no objeto `LunchCashierQueue`. O código a seguir declara `Queue` como uma classe base virtual:

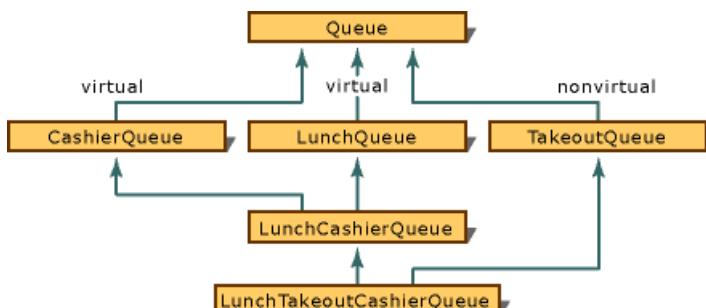
```
// deriv_VirtualBaseClasses.cpp
// compile with: /LD
class Queue {};
class CashierQueue : virtual public Queue {};
class LunchQueue : virtual public Queue {};
class LunchCashierQueue : public LunchQueue, public CashierQueue {};
```

A `virtual` palavra-chave garante que apenas uma cópia do subobjeto `Queue` seja incluída (consulte a figura a seguir).



Objeto de linha de almoço simulado com classes base virtuais

Uma classe pode ter um componente virtual e um componente não virtual de determinado tipo. Isso acontece nas condições ilustradas na figura a seguir.



Componentes virtuais e não virtuais da mesma classe

Na figura, `CashierQueue` e `LunchQueue` usam `Queue` como uma classe base virtual. No entanto, `TakeoutQueue` especifica `Queue` como uma classe base, não uma classe base virtual. Portanto, `LunchTakeoutCashierQueue` tem dois subobjetos do tipo `Queue`: um do caminho de herança que inclui `LunchCashierQueue` e outro do caminho que inclui `TakeoutQueue`. Isso é ilustrado na figura a seguir.



Layout de objeto com herança virtual e não virtual

NOTE

A herança virtual oferece benefícios significativos de tamanho quando comparada com a herança não virtual. No entanto, pode apresentar a sobrecarga adicional de processamento.

Se uma classe derivada substitui uma função virtual que herda de uma classe base virtual e, se um construtor ou um destruidor para a classe base derivada chamar essa função usando um ponteiro para a classe base virtual, o compilador virtual poderá inserir campos "vtordisp" adicionais ocultos nas classes com bases virtuais. A `/vd0` opção do compilador suprime a adição do membro de substituição do Construtor vtordisp/destruidor oculto. A `/vd1` opção do compilador, o padrão, habilita-os onde eles são necessários. Desative vtordisps apenas se você tiver certeza de que todos os destruidores e construtores da classe chamam funções virtuais virtualmente.

A `/vd` opção do compilador afeta um módulo de compilação inteiro. Use o `vtordisp` pragma para suprimir e reabilitar os `vtordisp` campos de acordo com a classe:

```

#pragma vtordisp( off )
class GetReal : virtual public { ... };
\n#pragma vtordisp( on )

```

Ambiguidades de nome

A herança múltipla apresenta a possibilidade de que os nomes sejam herdados ao longo de mais de um caminho. Os nomes de membros de classe ao longo desses caminhos não são necessariamente exclusivos. Esses conflitos de nome são chamados de "ambiguidades".

Qualquer expressão que se referir a um membro de classe deve fazer uma referência não ambígua. O exemplo a seguir mostra como as ambiguidades se desenvolvem:

```

// deriv_NameAmbiguities.cpp
// compile with: /LD
// Declare two base classes, A and B.
class A {
public:
    unsigned a;
    unsigned b();
};

class B {
public:
    unsigned a(); // Note that class A also has a member "a"
    int b(); // and a member "b".
    char c;
};

// Define class C as derived from A and B.
class C : public A, public B {};

```

Dadas as declarações de classe anteriores, códigos como o seguinte são ambíguos porque não está claro se `b` se refere a `b` em `A` ou em `B`:

```
C *pc = new C;
```

```
pc->b();
```

Considere o exemplo anterior. Como o nome `a` é um membro das classes `A` e `B`, o compilador não consegue distinguir qual `a` designa a função a ser chamada. O acesso a um membro é ambíguo se pode referenciar mais de uma função, objeto, tipo ou enumerador.

O compilador detecta ambiguidades executando testes nesta ordem:

1. Se o acesso ao nome é ambíguo (como somente descrito), será gerada uma mensagem de erro.
2. Se as funções sobrecarregadas são inequívocas, elas são resolvidas.
3. Se o acesso ao nome viola a permissão de acesso a membros, será gerada uma mensagem de erro. (Para obter mais informações, consulte [controle de acesso de membro](#).)

Quando uma expressão gera uma ambiguidade por meio de uma herança, você pode solucioná-la manualmente qualificando o nome em questão com seu nome de classe. Para fazer com que o exemplo acima seja compilado corretamente sem ambiguidades, use códigos como:

```
C *pc = new C;
```

```
pc->B::a();
```

NOTE

Quando `c` é declarado, ele tem o potencial para causar erros quando `B` é referenciado no escopo de `c`. Nenhum erro é emitido, no entanto, até que uma referência não qualificada a `B` seja feita no escopo de `c`.

Dominância

É possível que mais de um nome (função, objeto ou enumerador) seja acessado por meio de um gráfico de herança. Esses casos são considerados ambíguos com classes base não virtuais. Eles também são ambíguos com classes base virtuais, a menos que um dos nomes "domine" os outros.

Um nome domina outro nome quando é definido em ambas as classes e uma classe é derivada da outra. O nome dominante é o nome na classe derivada; esse nome é usado quando uma ambiguidade poderia surgir de outra forma, conforme mostrado no seguinte exemplo:

```

// deriv_Dominance.cpp
// compile with: /LD
class A {
public:
    int a;
};

class B : public virtual A {
public:
    int a();
};

class C : public virtual A {};

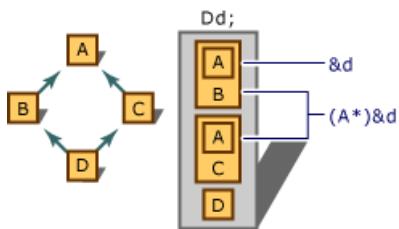
class D : public B, public C {
public:
    D() { a(); } // Not ambiguous. B::a() dominates A::a.
};

```

Conversões ambíguas

As conversões explícitas e implícitas de ponteiros ou as referências aos tipos da classe podem causar ambiguidades. A figura a seguir, a conversão ambígua dos ponteiros para as classes base, mostra o seguinte:

- A declaração de um objeto de tipo `D`.
- O efeito de aplicar o address-of Operator (`&`) a esse objeto. Observe que o operador address-of sempre fornece o endereço base do objeto.
- O efeito de converter explicitamente o ponteiro obtido usando o operador address-of para o tipo de classe base `A`. Observe que a coerção do endereço do objeto para o tipo `A*` não fornece sempre ao compilador informações suficientes sobre o subobjeto do tipo `A` a ser selecionado; nesse caso, existem dois subobjetos.



Conversão ambígua de ponteiros para classes base

A conversão para o tipo `A*` (ponteiro para `A`) é ambígua porque não há como distinguir qual subobjeto do tipo `A` está correto. Observe que você pode evitar a ambiguidade explicitamente especificando qual subobjeto você pretende usar, como segue:

```

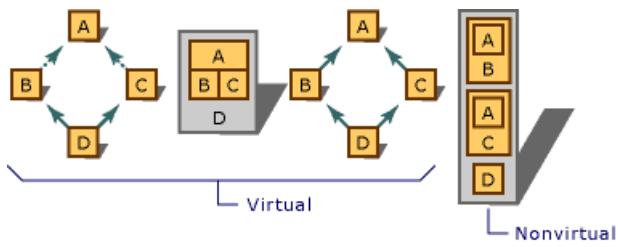
(A *)(B *)&d      // Use B subobject.
(A *)(C *)&d      // Use C subobject.

```

Ambiguidades e classes base virtuais

Se forem usadas classes base virtuais, as funções, os objetos, os tipos e os enumeradores poderão ser alcançados por meio de caminhos de herança múltipla. Como há somente uma instância da classe base, não há nenhuma ambiguidade ao acessar esses nomes.

A figura a seguir mostra como os objetos são compostos usando a herança virtual e não virtual.



Derivação virtual versus não virtual

Na figura, o acesso a qualquer membro da classe **A** por meio de classes base não virtuais causa uma ambiguidade; o compilador não tem nenhuma informação que explique se ele deve usar o subobjeto associado a **B** ou o subobjeto associado a **C**. No entanto, quando **A** é especificada como uma classe base virtual, não há dúvida sobre qual subobjeto está sendo acessado.

Confira também

[Herança](#)

Substituições Explícitas (C++)

25/03/2020 • 2 minutes to read • [Edit Online](#)

Seção específica da Microsoft

Se a mesma função virtual for declarada em duas ou mais interfaces e se uma classe for derivada dessas interfaces, você poderá substituir explicitamente cada função virtual.

Para obter informações sobre substituições explícitas em C++ código gerenciado usando/CLI, consulte [substituições explícitas](#).

Fim da seção específica da Microsoft

Exemplo

O exemplo de código a seguir ilustra como usar as substituições explícitas:

```
// deriv_ExplicitOverrides.cpp
// compile with: /GR
extern "C" int printf_s(const char *, ...);

__interface IMyInt1 {
    void mf1();
    void mf1(int);
    void mf2();
    void mf2(int);
};

__interface IMyInt2 {
    void mf1();
    void mf1(int);
    void mf2();
    void mf2(int);
};

class CMyClass : public IMyInt1, public IMyInt2 {
public:
    void IMyInt1::mf1() {
        printf_s("In CMyClass::IMyInt1::mf1()\n");
    }

    void IMyInt1::mf1(int) {
        printf_s("In CMyClass::IMyInt1::mf1(int)\n");
    }

    void IMyInt1::mf2();
    void IMyInt1::mf2(int);

    void IMyInt2::mf1() {
        printf_s("In CMyClass::IMyInt2::mf1()\n");
    }

    void IMyInt2::mf1(int) {
        printf_s("In CMyClass::IMyInt2::mf1(int)\n");
    }

    void IMyInt2::mf2();
    void IMyInt2::mf2(int);
};
```

```

void CMyClass::IMyInt1::mf2() {
    printf_s("In CMyClass::IMyInt1::mf2()\n");
}

void CMyClass::IMyInt1::mf2(int) {
    printf_s("In CMyClass::IMyInt1::mf2(int)\n");
}

void CMyClass::IMyInt2::mf2() {
    printf_s("In CMyClass::IMyInt2::mf2()\n");
}

void CMyClass::IMyInt2::mf2(int) {
    printf_s("In CMyClass::IMyInt2::mf2(int)\n");
}

int main() {
    IMyInt1 *pIMyInt1 = new CMyClass();
    IMyInt2 *pIMyInt2 = dynamic_cast<IMyInt2 *>(pIMyInt1);

    pIMyInt1->mf1();
    pIMyInt1->mf1(1);
    pIMyInt1->mf2();
    pIMyInt1->mf2(2);
    pIMyInt2->mf1();
    pIMyInt2->mf1(3);
    pIMyInt2->mf2();
    pIMyInt2->mf2(4);

    // Cast to a CMyClass pointer so that the destructor gets called
    CMyClass *p = dynamic_cast<CMyClass *>(pIMyInt1);
    delete p;
}

```

```

In CMyClass::IMyInt1::mf1()
In CMyClass::IMyInt1::mf1(int)
In CMyClass::IMyInt1::mf2()
In CMyClass::IMyInt1::mf2(int)
In CMyClass::IMyInt2::mf1()
In CMyClass::IMyInt2::mf1(int)
In CMyClass::IMyInt2::mf2()
In CMyClass::IMyInt2::mf2(int)

```

Confira também

[Herança](#)

Classes abstratas (C++)

25/03/2020 • 3 minutes to read • [Edit Online](#)

As classes abstratas agem como expressões de conceitos gerais das quais classes mais específicas podem ser derivadas. Não é possível criar um objeto de um tipo de classe abstrata; no entanto, é possível usar ponteiros e referências para tipos de classes abstratas.

Uma classe que contenha ao menos uma função virtual pura é considerada uma classe abstrata. Classes derivadas da classe abstrata devem implementar a função virtual pura, ou serão também classes abstratas.

Considere o exemplo apresentado em [funções virtuais](#). A intenção da classe `Account` é fornecer a funcionalidade geral, mas os objetos do tipo `Account` são muito gerais para serem úteis. Portanto, `Account` é um bom candidato para uma classe abstrata:

```
// deriv_AbstractClasses.cpp
// compile with: /LD
class Account {
public:
    Account( double d );    // Constructor.
    virtual double GetBalance();    // Obtain balance.
    virtual void PrintBalance() = 0;    // Pure virtual function.
private:
    double _balance;
};
```

A única diferença entre essa declaração e a anterior é que `PrintBalance` é declarado com o especificador puro (`= 0`).

Restrições em classes abstratas

As classes abstratas não podem ser usadas para:

- Variáveis ou dados de membro
- Tipos de argumento
- Tipos de retorno de função
- Tipos de conversões explícitas

Outra restrição é que se o construtor para uma classe abstrata chamar uma função virtual pura, direta ou indiretamente, o resultado será indefinido. No entanto, os construtores e os destruidores para classes abstratas podem chamar outras funções de membro.

As funções virtuais puras podem ser definidas para classes abstratas, mas podem ser chamadas diretamente somente usando a sintaxe:

nome de classe abstrata:Function-Name()

Isso ajuda a criar hierarquias de classes cujas classes base incluem destruidores virtuais puros, porque os destruidores de classe base sempre são chamados no processo de destruição de um objeto. Considere o exemplo a seguir:

```
// Declare an abstract base class with a pure virtual destructor.  
// deriv_RestrictionsOnUsingAbstractClasses.cpp  
class base {  
public:  
    base() {}  
    virtual ~base()=0;  
};  
  
// Provide a definition for destructor.  
base::~base() {}  
  
class derived:public base {  
public:  
    derived() {}  
    ~derived(){}  
};  
  
int main() {  
    derived *pDerived = new derived;  
    delete pDerived;  
}
```

Quando o objeto apontado por `pDerived` é excluído, o destruidor da classe `derived` é chamado e o destruidor da classe `base` é chamado. A implementação vazia para a função virtual pura garante que pelo menos uma implementação exista para a função.

NOTE

No exemplo anterior, a função virtual pura `base::~base` é chamada implicitamente de `derived::~derived`. Também é possível chamar funções virtuais puras explicitamente usando um nome de função de membro totalmente qualificado.

Confira também

[Herança](#)

Resumo das regras de escopo

02/09/2020 • 6 minutes to read • [Edit Online](#)

O uso de um nome deve ser inequívoco dentro de seu escopo (até o ponto onde a sobrecarga é determinada). Se o nome denota uma função, a função deve ser inequívoca em relação ao número e ao tipo de parâmetros. Se o nome permanecer inequívoco, as regras [de acesso de membro](#) serão aplicadas.

Inicializadores de construtores

[Inicializadores de Construtor](#) são avaliados no escopo do bloco mais externo do construtor para o qual eles são especificados. Portanto, eles podem usar os nomes de parâmetro do construtor.

Nomes globais

Um nome de um objeto, função ou enumerador é global se introduzido fora de qualquer função ou classe, ou se prefixado pelo operador unário global de escopo (`::`), além de não usado junto a qualquer um desses operadores binários:

- Resolução do escopo (`::`)
- Seleção de membros para objetos e referências (`.`)
- Seleção de membros para ponteiros (`->`)

Nomes qualificados

Os nomes usados com o operador de resolução de escopo binário (`::`) são chamados de "nomes qualificados". O nome especificado depois do operador de resolução de escopo deve ser membro da classe especificada à esquerda do operador ou membro de sua classe base.

Nomes especificados após o operador de seleção de membro (`.` or `->`) devem ser membros do tipo de classe do objeto especificado à esquerda do operador ou membros de sua classe base (es). Os nomes especificados à direita do operador de seleção de membro (`->`) também podem ser objetos de outro tipo de classe, desde que o lado esquerdo de `->` seja um objeto de classe e que a classe defina um operador de seleção de membro sobrecarregado (`->`) que seja avaliado como um ponteiro para algum outro tipo de classe. (Esse provisionamento é discutido em mais detalhes em [acesso de membro de classe](#).)

O compilador pesquisa por nomes na seguinte ordem, parando quando o nome é encontrado:

1. Escopo do bloco atual se o nome for usado em uma função; caso contrário, o escopo global.
2. Fora de cada escopo de bloco delimitador, incluindo o escopo de função mais externo (que inclui parâmetros de função).
3. Se o nome for usado dentro de uma função de membro, o escopo da classe será pesquisado por nome.
4. As classes base da classe são pesquisadas sequencialmente pelo nome.
5. O escopo delimitador da classe aninhada (se houver) e suas bases são pesquisados. A pesquisa continua até que o escopo delimitador de classe externo ser pesquisado.
6. O escopo global é pesquisado.

No entanto, você pode modificar essa ordem de pesquisa como segue:

1. Os nomes precedidos por `::` forçam a pesquisa a iniciar no escopo global.
2. Os nomes precedidos `class` pelas `struct` `union` palavras-chave, e forçam o compilador a pesquisar somente `class` `struct` nomes, ou `union`.
3. Os nomes no lado esquerdo do operador de resolução de escopo (`::`) só podem ser `class` nomes,, `struct` `namespace` ou `union`.

Se o nome se refere a um membro não estático, mas é usado em uma função de membro estático, uma mensagem de erro é gerada. Da mesma forma, se o nome se referir a qualquer membro não estático em uma classe delimitadora, uma mensagem de erro será gerada porque as classes embutidas não têm ponteiros de classe delimitadores `this`.

Nomes de parâmetro de função

Os nomes de parâmetro de função nas definições de função são considerados no escopo do bloco mais externo da função. Portanto, eles são nomes locais e saem do escopo quando a função é encerrada.

Os nomes de parâmetro de função em declarações de função (protótipos) estão no escopo local da declaração e saem do escopo no final da declaração.

Os parâmetros padrão estão no escopo do parâmetro para o qual eles são o padrão, conforme descrito nos dois parágrafos anteriores. No entanto, eles não podem acessar variáveis locais nem membros de classes não estáticas. Os parâmetros padrão são avaliados no ponto da chamada de função, mas são avaliados no escopo original da declaração de função. Portanto, os parâmetros padrão para as funções de membro são sempre avaliados no escopo de classe.

Confira também

[Herança](#)

Palavras-chave de herança

02/09/2020 • 3 minutes to read • [Edit Online](#)

Específico da Microsoft

```
class [__single_inheritance] class-name;
class [__multiple_inheritance] class-name;
class [__virtual_inheritance] class-name;
```

onde:

nome da classe

O nome da classe que está sendo declarada.

O C++ permite que você declare um ponteiro para um membro de classe antes da definição da classe. Por exemplo:

```
class S;
int S::*p;
```

No código acima, `p` é declarado como um ponteiro para membro inteiro da classe S. No entanto, `class S` ainda não foi definido neste código; ele só foi declarado. Quando o compilador encontrar esse ponteiro, ele fará uma representação generalizada do ponteiro. O tamanho de representação depende do modelo de herança especificado. Há quatro maneiras de especificar um modelo de herança para o compilador:

- No IDE sob **representação de ponteiro para membro**
- Na linha de comando usando a opção [/V/MG](#)
- Usando o [pointers_to_members](#) pragma
- Usando as palavras-chave de herança `__single_inheritance`, `__multiple_inheritance` e `__virtual_inheritance`. Essa técnica controla o modelo de herança com base em classes.

NOTE

Se você sempre declara um ponteiro para um membro de uma classe depois de defini-la, você não precisa usar qualquer uma dessas opções.

A declaração de um ponteiro para um membro de uma classe antes da definição de classe afeta o tamanho e a velocidade do arquivo executável resultante. Quanto mais complexa a herança usada por uma classe, maior é o número de bytes necessários para representar um ponteiro para um membro da classe e maior é o código necessário interpretar o ponteiro. A herança única é a menos complexa, e a herança virtual é a mais complexa.

Se o exemplo anterior for alterado para:

```
class __single_inheritance S;
int S::*p;
```

independentemente das opções de linha de comando ou dos pragmas, os ponteiros para os membros de `class S` usarão a menor representação possível.

NOTE

A mesma declaração de encaminhamento de uma representação de ponteiro para membro de classe deve ocorrer em cada unidade de tradução que declarar ponteiros para membros daquela classe, e a declaração deve ocorrer antes que os ponteiros para os membros sejam declarados.

Para compatibilidade com versões anteriores, `_single_inheritance`, `_multiple_inheritance`, `_virtual_inheritance` são sinônimos para `_single_inheritance`, `_multiple_inheritance` e `_virtual_inheritance` a menos que a opção do compilador [/za \(desabilitar extensões de idioma\)](#) seja especificada.

FINAL específico da Microsoft

Confira também

[Palavras-chave](#)

virtual (C++)

02/09/2020 • 2 minutes to read • [Edit Online](#)

A `virtual` palavra-chave declara uma função virtual ou uma classe base virtual.

Sintaxe

```
virtual [type-specifiers] member-function-declarator  
virtual [access-specifier] base-class-name
```

parâmetros

especificadores de tipo

Especifica o tipo de retorno da função membro virtual.

função membro-Declarador

Declara uma função membro.

especificador de acesso

Define o nível de acesso à classe base, `public` `protected` ou `private`. Pode aparecer antes ou depois da `virtual` palavra-chave.

nome da classe base

Identifica um tipo de classe declarado previamente.

Comentários

Consulte [funções virtuais](#) para obter mais informações.

Consulte também as seguintes palavras-chave: [classe](#), [privada](#), [pública](#) e [protegida](#).

Confira também

Palavras-chave

Específico da Microsoft

Permite que você indique explicitamente que está chamando uma implementação da classe base para uma função que está substituindo.

Sintaxe

```
__super::member_function();
```

Comentários

Todos os métodos acessíveis da classe base são considerados durante a fase de resolução de sobrecarga e a função que fornece a melhor correspondência é a chamada.

__super Só pode aparecer dentro do corpo de uma função de membro.

__super Não pode ser usado com uma declaração using. Consulte [usando a declaração](#) para obter mais informações.

Com a introdução de [atributos](#) que injetam código, seu código pode conter uma ou mais classes base cujos nomes você talvez não conheça, mas que contêm métodos que você deseja chamar.

Exemplo

```
// deriv_super.cpp
// compile with: /c
struct B1 {
    void mf(int) {}
};

struct B2 {
    void mf(short) {}

    void mf(char) {}
};

struct D : B1, B2 {
    void mf(short) {
        __super::mf(1);    // Calls B1::mf(int)
        __super::mf('s');  // Calls B2::mf(char)
    }
};
```

FINAL específico da Microsoft

Confira também

[Palavras-chave](#)

__interface

02/09/2020 • 2 minutes to read • [Edit Online](#)

Específico da Microsoft

Uma interface do Microsoft C++ pode ser definida da seguinte maneira:

- Ela pode herdar de zero ou mais interfaces base.
- Ela não pode herdar de uma classe base.
- Ela pode conter apenas métodos virtuais puros e públicos.
- Ela não pode conter construtores, destruidores ou operadores.
- Ela não pode conter métodos estáticos.
- Ela não pode conter membros de dados; as propriedades são permitidas.

Sintaxe

```
modifier __interface interface-name {interface-definition};
```

Comentários

Uma [classe](#) ou [struct](#) do C++ poderia ser implementada com essas regras, mas as `__interface` impõe.

Por exemplo, o seguinte é uma definição de interface de exemplo:

```
__interface IMyInterface {
    HRESULT CommitX();
    HRESULT get_X(BSTR* pbstrName);
};
```

Para obter informações sobre interfaces gerenciadas, consulte [classe de interface](#).

Observe que você não precisa indicar explicitamente que as funções `CommitX` e `get_X` são puramente virtuais. Uma declaração equivalente para a primeira função seria:

```
virtual HRESULT CommitX() = 0;
```

`__interface` implica o modificador [novertable](#) `__declspec`.

Exemplo

O exemplo a seguir mostra como usar as propriedades declaradas em uma interface.

```
// deriv_interface.cpp
#define _ATL_ATTRIBUTES 1
#include <atlbases.h>
#include <atlcom.h>
#include <string.h>
#include <comdef.h>
```

```
#include <stdio.h>

[module(name="test")];

[ object, uuid("00000000-0000-0000-0000-000000000001"), library_block ]
__interface IFace {
    [ id(0) ] int int_data;
    [ id(5) ] BSTR bstr_data;
};

[ coclass, uuid("00000000-0000-0000-0000-000000000002") ]
class MyClass : public IFace {
private:
    int m_i;
    BSTR m_bstr;

public:
    MyClass()
    {
        m_i = 0;
        m_bstr = 0;
    }

    ~MyClass()
    {
        if (m_bstr)
            ::SysFreeString(m_bstr);
    }

    int get_int_data()
    {
        return m_i;
    }

    void put_int_data(int _i)
    {
        m_i = _i;
    }

    BSTR get_bstr_data()
    {
        BSTR bstr = ::SysAllocString(m_bstr);
        return bstr;
    }

    void put_bstr_data(BSTR bstr)
    {
        if (m_bstr)
            ::SysFreeString(m_bstr);
        m_bstr = ::SysAllocString(bstr);
    }
};

int main()
{
    _bstr_t bstr("Testing");
    CoInitialize(NULL);
    CComObject<MyClass>* p;
    CComObject<MyClass>::CreateInstance(&p);
    p->int_data = 100;
    printf_s("p->int_data = %d\n", p->int_data);
    p->bstr_data = bstr;
    printf_s("bstr_data = %S\n", p->bstr_data);
}
```

```
p->int_data = 100  
bstr_data = Testing
```

FINAL específico da Microsoft

Confira também

[Palavras-chave](#)

[Atributos de interface](#)

Funções de membro especiais

25/03/2020 • 3 minutes to read • [Edit Online](#)

As *funções de membro especiais* são funções de membro de classe (ou struct) que, em determinados casos, o compilador gera automaticamente para você. Essas funções são o [construtor padrão](#), o [destruidor](#), o [Construtor de cópia e o operador de atribuição de cópia](#) e o [Construtor de movimentação e o operador de atribuição de movimento](#). Se sua classe não definir uma ou mais das funções de membro especiais, o compilador poderá declarar e definir implicitamente as funções que são usadas. As implementações geradas pelo compilador são chamadas de funções de membro especiais *padrão*. O compilador não gerará funções se elas não forem necessárias.

Você pode declarar explicitamente uma função de membro especial padrão usando a palavra-chave = **Default** . Isso faz com que o compilador defina a função somente se necessário, da mesma maneira como se a função não fosse declarada.

Em alguns casos, o compilador pode gerar funções de membro especiais *excluídas* , que não são definidas e, portanto, não podem ser chamadas. Isso pode acontecer em casos em que uma chamada para uma função de membro especial específica em uma classe não faz sentido, dadas outras propriedades da classe. Para impedir explicitamente a geração automática de uma função de membro especial, você pode declará-la como excluída usando a palavra-chave = **delete** .

O compilador gera um *construtor padrão*, um construtor que não usa argumentos, somente quando você não declarou nenhum outro construtor. Se você declarou apenas um construtor que usa parâmetros, o código que tenta chamar um construtor padrão faz com que o compilador produza uma mensagem de erro. O construtor padrão gerado pelo compilador executa uma [inicialização padrão](#) simples de membro do objeto. A inicialização padrão deixa todas as variáveis de membro em um estado indeterminado.

O destruidor padrão executa a destruição de membros do objeto. Ele será virtual somente se um destruidor de classe base for virtual.

As operações de construção e atribuição de cópia e movimentação padrão executam cópias de padrão de bits ou movimentações de membros de dados não estáticos. As operações de movimentação são geradas somente quando nenhuma operação de destruidor ou de movimentação ou cópia é declarada. Um construtor de cópia padrão só é gerado quando nenhum construtor de cópia é declarado. Ele será excluído implicitamente se uma operação de movimentação for declarada. Um operador de atribuição de cópia padrão é gerado somente quando nenhum operador de atribuição de cópia é declarado explicitamente. Ele será excluído implicitamente se uma operação de movimentação for declarada.

Confira também

[Referência da linguagem C++](#)

Membros estáticos (C++)

02/09/2020 • 3 minutes to read • [Edit Online](#)

As classes podem conter dados estáticos de membros e funções de membros. Quando um membro de dados é declarado como `static`, apenas uma cópia dos dados é mantida para todos os objetos da classe.

Os membros de dados estáticos não fazem parte dos objetos de um determinado tipo de classe. Como resultado, a declaração de um membro de dados estáticos não é considerada uma definição. O membro de dados é declarado no escopo da classe, mas a definição é realizada no escopo do arquivo. Esses membros estáticos têm vinculação externa. O exemplo a seguir ilustra isso:

```
// static_data_members.cpp
class BufferedOutput
{
public:
    // Return number of bytes written by any object of this class.
    short BytesWritten()
    {
        return bytecount;
    }

    // Reset the counter.
    static void ResetCount()
    {
        bytecount = 0;
    }

    // Static member declaration.
    static long bytecount;
};

// Define bytecount in file scope.
long BufferedOutput::bytecount;

int main()
{
}
```

No código anterior, o membro `bytecount` é declarado na classe `BufferedOutput`, mas deve ser definido fora de declaração da classe.

Os membros de dados estáticos podem ser mencionados sem fazer referência a um objeto do tipo da classe. O número de bytes gravados usando objetos `BufferedOutput` pode ser obtido da seguinte maneira:

```
long nBytes = BufferedOutput::bytecount;
```

Para que o membro estático exista, não é necessário que objetos do tipo de classe existam. Membros estáticos também podem ser acessados usando a seleção de Membros (`. -> operadores and`). Por exemplo:

```
BufferedOutput Console;

long nBytes = Console.bytecount;
```

No caso anterior, a referência ao objeto (`Console`) não é avaliada. O valor retornado é o do objeto estático

`bytecount` .

Os membros de dados estáticos estão sujeitos às regras de acesso do membro da classe. Portanto, o acesso particular aos membros de dados estáticos é permitido somente para funções de membro da classe e friends. Essas regras são descritas em [controle de acesso de membro](#). A exceção é que os membros de dados estáticos devem ser definidos no escopo de arquivo, independentemente das suas restrições de acesso. Se o membro de dados for inicializado explicitamente, um inicializador deverá ser fornecido com a definição.

O tipo de um membro estático não é qualificado por seu nome de classe. Portanto, o tipo de

`BufferedOutput::bytecount` é `long` .

Confira também

[Classes e structs](#)

Classes C++ como tipos de valor

02/12/2019 • 6 minutes to read • [Edit Online](#)

C++ as classes são por tipos de valor padrão. Eles podem ser especificados como tipos de referência, o que permite o comportamento polimórfico para dar suporte à programação orientada a objeto. Os tipos de valor às vezes são exibidos a partir da perspectiva da memória e do controle de layout, enquanto os tipos de referência são sobre classes base e funções virtuais para fins polimórficos. Por padrão, os tipos de valor são copiáveis, o que significa que há sempre um construtor de cópia e um operador de atribuição de cópia. Para tipos de referência, você torna a classe não-copiável (desabilita o construtor de cópia e o operador de atribuição de cópia) e usa um destruidor virtual que dá suporte ao polimorfismo pretendido. Os tipos de valor também são sobre o conteúdo, que, quando eles são copiados, sempre fornecem dois valores independentes que podem ser modificados separadamente. Tipos de referência são sobre identidade – que tipo de objeto é? Por esse motivo, "tipos de referência" também são chamados de "tipos polimórficos".

Se você realmente quiser um tipo de referência (classe base, funções virtuais), será necessário desabilitar explicitamente a cópia, conforme mostrado na classe `MyRefType` no código a seguir.

```
// cl /EHsc /nologo /W4

class MyRefType {
private:
    MyRefType & operator=(const MyRefType &);
    MyRefType(const MyRefType &);

public:
    MyRefType () {}
};

int main()
{
    MyRefType Data1, Data2;
    // ...
    Data1 = Data2;
}
```

A compilação do código acima resultará no seguinte erro:

```
test.cpp(15) : error C2248: 'MyRefType::operator =' : cannot access private member declared in class
'MyRefType'
    meow.cpp(5) : see declaration of 'MyRefType::operator ='
    meow.cpp(3) : see declaration of 'MyRefType'
```

Tipos de valor e eficiência de movimentação

A sobrecarga de alocação de cópia é evitada devido a novas otimizações de cópia. Por exemplo, quando você insere uma cadeia de caracteres no meio de um vetor de cadeias de caracteres, não haverá sobrecarga de realocação de cópia, apenas uma mudança, mesmo se resultar em um crescimento do próprio vetor. Isso também se aplica a outras operações, por exemplo, executando uma operação de adição em dois objetos muito grandes. Como você habilita essas otimizações de operação de valor? Em alguns C++ compiladores, o compilador permitirá isso para você implicitamente, assim como os construtores de cópia podem ser gerados automaticamente pelo compilador. No entanto C++, no, sua classe precisará "aceitar" para mover a atribuição e os construtores declarando-os em sua definição de classe. Isso é feito usando a referência a "e comercial dupla" (& &) rvalue nas declarações de função de membro apropriadas e definindo os métodos de atribuição de construtor

e de movimentação de movimentação. Você também precisa inserir o código correto para "roubar o entradas" do objeto de origem.

Como você decide se precisa de uma movimentação habilitada? Se você já sabe que precisa de construção de cópia habilitada, provavelmente deseja mover habilitada se ela puder ser mais barata do que uma cópia em profundidade. No entanto, se você souber que precisa de suporte de movimentação, isso não significa necessariamente que você deseja que a cópia seja habilitada. Esse último caso seria chamado de "tipo somente de movimentação". Um exemplo já na biblioteca padrão é `unique_ptr`. Como uma observação adicional, o antigo `auto_ptr` é preferido e foi substituído por `unique_ptr` precisamente devido à falta de suporte à semântica de movimentação na versão anterior do C++.

Usando a semântica de movimentação, você pode retornar por valor ou inserir no meio. Move é uma otimização da cópia. Há necessidade de alocação de heap como uma solução alternativa. Considere o seguinte pseudocódigo:

```
#include <set>
#include <vector>
#include <string>
using namespace std;

//...
set<widget> LoadHugeData() {
    set<widget> ret;
    // ... load data from disk and populate ret
    return ret;
}
//...
widgets = LoadHugeData(); // efficient, no deep copy

vector<string> v = IfIHadAMillionStrings();
v.insert( begin(v)+v.size()/2, "scott" ); // efficient, no deep copy-shuffle
v.insert( begin(v)+v.size()/2, "Andrei" ); // (just 1M ptr/len assignments)
//...
HugeMatrix operator+(const HugeMatrix& , const HugeMatrix& );
HugeMatrix operator+(const HugeMatrix& , HugeMatrix&& );
HugeMatrix operator+(      HugeMatrix&& , const HugeMatrix& );
HugeMatrix operator+(      HugeMatrix&& , HugeMatrix&& );
//...
hm5 = hm1+hm2+hm3+hm4+hm5; // efficient, no extra copies
```

Habilitando a movimentação para tipos de valor apropriados

Para uma classe de valor em que a movimentação pode ser mais barata do que uma cópia em profundidade, habilite a construção de movimentação e a atribuição de movimento para obter eficiência. Considere o seguinte pseudocódigo:

```
#include <memory>
#include <stdexcept>
using namespace std;
// ...
class my_class {
    unique_ptr<BigHugeData> data;
public:
    my_class( my_class& other )    // move construction
        : data( move( other.data ) ) { }
    my_class& operator=( my_class&& other )    // move assignment
    { data = move( other.data ); return *this; }
    // ...
    void method() {    // check (if appropriate)
        if( !data )
            throw std::runtime_error("RUNTIME ERROR: Insufficient resources!");
    }
};
```

Se você habilitar a construção/atribuição de cópia, habilite também a construção/atribuição de movimentação se ela puder ser mais barata do que uma cópia profunda.

Alguns tipos *que não são de valor* são somente para movimentação, como quando você não pode clonar um recurso, somente transferir a propriedade. Exemplo: `unique_ptr`.

Consulte também

[C++ sistema de tipos](#)

[Bem-vindo de volta para C++](#)

[Referência da linguagem C++](#)

[Biblioteca padrão C++](#)

Conversões de tipo definido pelo usuário (C++)

02/09/2020 • 20 minutes to read • [Edit Online](#)

Uma *conversão* produz um novo valor de algum tipo a partir de um valor de um tipo diferente. As *conversões padrão* são incorporadas à linguagem C++ e dão suporte a seus tipos internos, e você pode criar *conversões definidas pelo usuário* para executar conversões de, de ou entre os tipos definidos pelo usuário.

As conversões padrão realizam conversões entre tipos internos, entre ponteiros ou referências para tipos relacionados por herança, para os ponteiros void e nulo. Para obter mais informações, consulte [conversões padrão](#). As conversões definidas pelo usuário realizam conversões entre tipos definidos pelo usuário ou entre tipos definidos pelo usuário e tipos internos. Você pode implementá-los como [construtores de conversão](#) ou como [funções de conversão](#).

As conversões podem ser implícitas (quando as chamadas do programador de um tipo a ser convertido para outro, como em uma inicialização direta ou convertida) ou implícita (quando o programa ou a linguagem chama um tipo diferente daquele fornecido pelo programador).

As conversões implícitas são tentadas quando:

- Um argumento fornecido para uma função não tem o mesmo tipo do parâmetro correspondente.
- O valor retornado de uma função não tem o mesmo tipo do tipo de retorno de função.
- Uma expressão de inicializador não tem o mesmo tipo do objeto que está sendo inicializado.
- Uma expressão que controla uma instrução condicional, constructos de loop ou opção não tem o tipo de resultado necessário para controlá-la.
- Um operando fornecido para um operador não tem o mesmo tipo do parâmetro do operando correspondente. Para operadores internos, ambos os operandos devem ter o mesmo tipo e são convertidos para um tipo comum que pode representar ambos. Para obter mais informações, consulte [conversões padrão](#). Para operadores definidos por usuários, cada operando deve ter o mesmo tipo do parâmetro operando correspondente.

Quando uma conversão padrão não pode concluir uma conversão implícita, o compilador pode usar uma conversão definida pelo usuário, seguida opcionalmente por uma conversão padrão adicional, para completá-la.

Quando duas ou mais conversões definidas pelo usuário que executam a mesma conversão estão disponíveis em um site de conversão, a conversão é chamada de ambígua. Essas ambiguidades são um erro porque o compilador não pode determinar qual das conversões disponíveis deve escolher. No entanto, não é errado apenas definir várias maneira de realizar a mesma conversão porque o conjunto de conversões disponíveis pode ser diferente em locais diferentes no código-fonte, por exemplo, dependendo de quais arquivos de cabeçalho estão incluídos no arquivo fonte. Enquanto apenas uma conversão estiver disponível no local da conversão, não há ambiguidade. Conversões ambíguas podem ocorrer de várias maneiras, mas as mais comuns são:

- Herança múltipla. A conversão é definida em mais de uma classe base.
- Chamada de função ambígua. A conversão é definida como um construtor de conversão do tipo de destino e como uma função de conversão do tipo de fonte. Para obter mais informações, consulte [funções de conversão](#).

Geralmente, você pode resolver uma ambiguidade apenas por meio da qualificação do nome do tipo envolvido ou realizando uma conversão explícita para esclarecer sua intenção.

Ambos, construtores de conversão e funções de conversão obedecem regras de controle de acesso de membro, mas a acessibilidade das conversões é apenas considerada se e quando uma conversão ambígua puder ser determinada. Isso significa que uma conversão pode ser ambígua ainda que o nível de acesso de uma conversão concorrente impeça o seu uso. Para obter mais informações sobre acessibilidade de membro, consulte [controle de acesso de membro](#).

A palavra-chave e problemas explícitos com conversão implícita

Por padrão, ao criar uma conversão definida pelo usuário, o compilador pode usá-la para executar conversões implícitas. Algumas vezes pode ser que você queira isso, mas outras vezes as regras simples que guiam o compilador para fazer conversões implícitas pode levá-lo a aceitar códigos que não quer.

Um exemplo bem conhecido de uma conversão implícita que pode causar problemas é a conversão para `bool`. Há muitas razões pelas quais você pode querer criar um tipo de classe que possa ser usado em um contexto booliano — por exemplo, para que possa ser usado para controlar uma `if` instrução ou um loop — mas quando o compilador executa uma conversão definida pelo usuário em um tipo interno, o compilador tem permissão para aplicar uma conversão padrão adicional posteriormente. A intenção dessa conversão padrão adicional é permitir coisas como a promoção de `short` para `int`, mas também abre a porta para conversões menos óbvias — por exemplo, de `bool` para `int`, que permite que seu tipo de classe seja usado em contextos inteiros que você nunca pretendeu. Esse problema específico é conhecido como o *problema booliano seguro*. Esse tipo de problema é onde a `explicit` palavra-chave pode ajudar.

A `explicit` palavra-chave informa ao compilador que a conversão especificada não pode ser usada para executar conversões implícitas. Se você quisesse a conveniência sintática de conversões implícitas antes da `explicit` introdução da palavra-chave, era necessário aceitar as consequências involuntárias que às vezes a conversão implícita criava ou usar funções de conversão nomeadas menos convenientes como uma solução alternativa. Agora, ao usar a `explicit` palavra-chave, você pode criar conversões convenientes que só podem ser usadas para executar transmissões explícitas ou inicialização direta, e isso não levará a um tipo de problema exemplificado pelo problema booliano seguro.

A `explicit` palavra-chave pode ser aplicada aos construtores de conversão desde o C++ 98 e às funções de conversão desde o C++ 11. As seções a seguir contêm mais informações sobre como usar a `explicit` palavra-chave.

Construtores de conversão

Os construtores de conversão definem conversões de tipos internos ou definidos pelo usuário para um tipo definido pelo usuário. O exemplo a seguir demonstra um construtor de conversão que converte do tipo interno `double` para um tipo definido pelo usuário `Money`.

```

#include <iostream>

class Money
{
public:
    Money() : amount{ 0.0 } {};
    Money(double _amount) : amount{ _amount } {};

    double amount;
};

void display_balance(const Money balance)
{
    std::cout << "The balance is: " << balance.amount << std::endl;
}

int main(int argc, char* argv[])
{
    Money payable{ 79.99 };

    display_balance(payable);
    display_balance(49.95);
    display_balance(9.99f);

    return 0;
}

```

Observe que a primeira chamada para a função `display_balance`, que leva um argumento do tipo `Money`, não necessita de uma conversão porque seu argumento é do tipo correto. No entanto, na segunda chamada para `display_balance`, uma conversão é necessária porque o tipo do argumento, um `double` com um valor de `49.95`, não é o que a função espera. A função não pode usar esse valor diretamente, mas como há uma conversão do tipo do argumento — `double` — para o tipo do parâmetro correspondente — `Money` — um valor temporário do tipo `Money` é construído a partir do argumento e usado para concluir a chamada de função. Na terceira chamada para `display_balance`, observe que o argumento não é um `double`, mas, em vez disso, é um `float` com um valor de `9.99` — e, ainda assim, a chamada de função ainda pode ser concluída porque o compilador pode executar uma conversão padrão — nesse caso, de `float` para `double` — e, em seguida, executar a conversão definida pelo usuário de `double` para `Money` para concluir a conversão necessária.

Declaração de construtores de conversão

As regras a seguir aplicam-se à declaração de um construtor de conversão:

- O tipo de destino da conversão é o tipo definido pelo usuário que está sendo construído.
- Geralmente, os construtores de conversão usam exatamente um argumento, que é o do tipo do código-fonte. No entanto, um construtor de conversão pode especificar parâmetros adicionais, caso cada parâmetro adicional tenha um valor padrão. O tipo de código-fonte permanece o tipo do primeiro parâmetro.
- Os construtores de conversão, como todos os construtores, não especificam um tipo de retorno. Especificar um tipo de retorno na declaração é um erro.
- Os construtores de conversão podem ser explícitos.

Construtores de conversão explícita

Ao declarar um construtor de conversão para ser `explicit`, ele só pode ser usado para executar a inicialização direta de um objeto ou para executar uma conversão explícita. Isso evita que funções que aceitam um argumento do tipo de classe também aceitem argumentos implicitamente do tipo de fonte do construtor de conversão, evitando que o tipo de classe seja inicializado por cópia a partir de um valor do tipo de fonte. O exemplo a seguir demonstra como definir um construtor de conversão explícita e o efeito que tem sobre qual

código é bem formado.

```
#include <iostream>

class Money
{
public:
    Money() : amount{ 0.0 } {};
    explicit Money(double _amount) : amount{ _amount } {};

    double amount;
};

void display_balance(const Money balance)
{
    std::cout << "The balance is: " << balance.amount << std::endl;
}

int main(int argc, char* argv[])
{
    Money payable{ 79.99 };           // Legal: direct initialization is explicit.

    display_balance(payable);         // Legal: no conversion required
    display_balance(49.95);           // Error: no suitable conversion exists to convert from double to Money.
    display_balance((Money)9.99f);    // Legal: explicit cast to Money

    return 0;
}
```

Nesse exemplo, observe que você pode ainda usar o construtor de conversão explícita para executar a inicialização direta de `payable`. Se ao invés disso você tivesse que inicializar `Money payable = 79.99;` por cópia, o resultado seria um erro. A primeira chamada para `display_balance` é sem efeito porque o argumento é o tipo correto. A segunda chamada para `display_balance` é um erro, porque o construtor de conversão não pode ser usado para executar conversões implícitas. A terceira chamada para `display_balance` é legal devido à conversão explícita para `Money`, mas observe que o compilador ainda ajudou a concluir a conversão inserindo uma conversão implícita de `float` para `double`.

Embora a praticidade de permitir conversões implícitas possa ser tentadora, fazer isso pode introduzir bugs difíceis de encontrar. A regra recomendada é tornar explícitos todos os construtores de conversão, exceto quando você tem certeza que deseja que uma conversão específica ocorra implicitamente.

Funções de conversão

As funções de conversão definem conversões de um tipo definidas pelo usuário para outros tipos. Essas funções são algumas vezes chamadas como operadores cast pois, juntamente com os construtores de conversão, são chamadas quando um valor é convertido para um tipo diferente. O exemplo a seguir demonstra uma função de conversão que converte do tipo definido pelo usuário, `Money`, para um tipo interno, `double`:

```

#include <iostream>

class Money
{
public:
    Money() : amount{ 0.0 } {};
    Money(double _amount) : amount{ _amount } {};

    operator double() const { return amount; }
private:
    double amount;
};

void display_balance(const Money balance)
{
    std::cout << "The balance is: " << balance << std::endl;
}

```

Observe que a variável de membro `amount` é tornada privada e que uma função de conversão pública para Type `double` é introduzida apenas para retornar o valor de `amount`. Na função `display_balance`, uma conversão implícita ocorre quando o valor de `balance` é transmitido para saída padrão usando o operador de inserção de fluxo `<<`. Como nenhum operador de inserção de fluxo é definido para o tipo definido pelo usuário `Money`, mas há um para o tipo interno `double`, o compilador pode usar a função de conversão de `Money` para `double` para satisfazer o operador de inserção de fluxo.

As funções de conversão são herdadas por classes derivadas. As funções de conversão em uma classe derivada apenas substitui uma função de conversão herdada quando convertidas exatamente para o mesmo tipo. Por exemplo, uma função de conversão definida pelo usuário do operador de classe derivada `int` não substitui — ou até mesmo influencia — uma função de conversão definida pelo usuário do operador de classe base curta, embora as conversões padrão definam uma relação de conversão entre `int` e `short`.

Declaração de funções de conversão

As regras a seguir são aplicadas ao declarar uma função de conversão:

- O tipo de destino da conversão deve ser declarado antes da declaração da função de conversão. Classes, estruturas, enumerações e `typedefs` não podem ser declarados na declaração da função de conversão.

```
operator struct String { char string_storage; }() // illegal
```

- As funções de conversão não têm argumentos. Especificar quaisquer parâmetros na declaração é um erro.
- As funções de conversão têm um tipo de retorno que é especificado pelo nome da função de conversão, que é também o nome do tipo de destino da conversão. Especificar um tipo de retorno na declaração é um erro.
- As funções de conversão podem ser virtuais.
- As funções de conversão podem ser explícitas.

Funções de conversão explícitas

Quando uma função de conversão é declarada como explícita, ele pode ser usada para executar uma conversão explícita. Isso evita que funções que aceitam um argumento do tipo de destino da função de conversão também aceitem implicitamente argumentos do tipo de classe, além de evitar que o tipo de destino seja inicializado por cópia, a partir de um valor do tipo de classe. O exemplo a seguir demonstra como definir uma função de conversão explícita e o efeito que tem sobre qual código é bem formado.

```
#include <iostream>

class Money
{
public:
    Money() : amount{ 0.0 } {};
    Money(double _amount) : amount{ _amount } {};

    explicit operator double() const { return amount; }
private:
    double amount;
};

void display_balance(const Money balance)
{
    std::cout << "The balance is: " << (double)balance << std::endl;
}
```

Aqui, o **operador** de função de conversão Double foi explicitado e uma conversão explícita para `double` o tipo foi introduzida na função `display_balance` para executar a conversão. Se essa conversão for omitida, o compilador será incapaz de localizar um operador de inserção de fluxo adequado `<<` de tipo `Money` e poderá ocorrer um erro.

Membros de dados mutáveis (C++)

02/09/2020 • 2 minutes to read • [Edit Online](#)

Essa palavra-chave só pode ser aplicada aos membros de dados não estáticas e não constantes de uma classe. Se um membro de dados for declarado `mutable`, será legal atribuir um valor a esse membro de dados de uma `const` função de membro.

Sintaxe

```
mutable member-variable-declaration;
```

Comentários

Por exemplo, o código a seguir será compilado sem erro porque foi `m_accessCount` declarado como sendo `mutable` e, portanto, pode ser modificado por `GetFlag` embora `GetFlag` seja uma função de membro `const`.

```
// mutable.cpp
class X
{
public:
    bool GetFlag() const
    {
        m_accessCount++;
        return m_flag;
    }
private:
    bool m_flag;
    mutable int m_accessCount;
};

int main()
{
```

Confira também

[Palavras-chave](#)

Declarações de classe aninhada

02/09/2020 • 7 minutes to read • [Edit Online](#)

Uma classe pode ser declarada dentro do escopo de outra classe. Essa classe é chamada de "classe aninhada". As classes aninhadas são consideradas como estando dentro do escopo da classe delimitadora e estão disponíveis para uso dentro desse escopo. Para fazer referência a uma classe aninhada a partir de um escopo diferente de seu escopo delimitador imediato, você deve usar um nome totalmente qualificado.

O exemplo a seguir mostra como declarar as classes aninhadas:

```
// nested_class_declarations.cpp
class BufferedIO
{
public:
    enum IOError { None, Access, General };

    // Declare nested class BufferedInput.
    class BufferedInput
    {
    public:
        int read();
        int good()
        {
            return _inputerror == None;
        }
    private:
        IOError _inputerror;
    };
};

// Declare nested class BufferedOutput.
class BufferedOutput
{
    // Member list
};
};

int main()
{
}
```

`BufferedIO::BufferedInput` e `BufferedIO::BufferedOutput` são declarados em `BufferedIO`. Esses nomes de classe não são visíveis fora do escopo da classe `BufferedIO`. No entanto, um objeto do tipo `BufferedIO` não contém nenhum objetos dos tipos `BufferedInput` ou `BufferedOutput`.

As classes aninhadas podem usar diretamente nomes, nomes de tipo, nomes de membros estáticos e enumeradores apenas da classe delimitadora. Para usar nomes de outros membros de classe, você deve usar ponteiros, referências ou nomes de objeto.

No exemplo de `BufferedIO` anterior, a enumeração `IOError` pode ser acessada diretamente por funções membro nas classes aninhadas, `BufferedIO::BufferedInput` ou `BufferedIO::BufferedOutput`, conforme mostrado na função `good`.

NOTE

As classes aninhadas declaram apenas os tipos dentro do escopo da classe. Elas não causam a criação de objetos contidos da classe aninhada. O exemplo anterior declara duas classes aninhadas, mas não declara objetos desses tipos de classe.

Uma exceção à visibilidade do escopo de uma declaração de classe aninhada é quando num nome de tipo é declarado junto com uma declaração de encaminhamento. Nesse caso, o nome da classe declarada pela declaração de encaminhamento é visível fora da classe delimitadora, com seu escopo definido como o menor escopo delimitador que não seja da classe. Por exemplo:

```
// nested_class_declarations_2.cpp
class C
{
public:
    typedef class U u_t; // class U visible outside class C scope
    typedef class V {} v_t; // class V not visible outside class C
};

int main()
{
    // okay, forward declaration used above so file scope is used
    U* pu;

    // error, type name only exists in class C scope
    u_t* pu2; // C2065

    // error, class defined above so class C scope
    V* pv; // C2065

    // okay, fully qualified name
    C::V* pv2;
}
```

Privilégio de acesso em classes aninhadas

O aninhamento de uma classe dentro de outra não concede privilégios de acesso especiais às funções membro da classe aninhada. Da mesma forma, as funções membro da classe delimitadora não têm acesso especial a membros da classe aninhada.

Funções de membro em classes aninhadas

As funções de membro declaradas em classes aninhadas podem ser definidas no escopo do arquivo. O exemplo anterior poderia ter sido escrito:

```

// member_functions_in_nested_classes.cpp
class BufferedIO
{
public:
    enum IOError { None, Access, General };
    class BufferedInput
    {
public:
    int read(); // Declare but do not define member
    int good(); // functions read and good.
private:
    IOError _inputerror;
};

class BufferedOutput
{
    // Member list.
};

};

// Define member functions read and good in
// file scope.
int BufferedIO::BufferedInput::read()
{
    return(1);
}

int BufferedIO::BufferedInput::good()
{
    return _inputerror == None;
}

int main()
{
}

```

No exemplo anterior, a sintaxe de *nome de tipo qualificado* é usada para declarar o nome da função. Esta declaração:

```
BufferedIO::BufferedInput::read()
```

significa que "a função `read`, que é um membro da classe `BufferedInput` e que está no escopo da classe `BufferedIO`". Como essa declaração usa a sintaxe de *nome de tipo qualificado*, as construções do seguinte formulário são possíveis:

```

typedef BufferedIO::BufferedInput BIO_INPUT;
int BIO_INPUT::read()

```

A declaração anterior é equivalente à anterior, mas usa um `typedef` nome no lugar dos nomes de classe.

Funções Friend em classes aninhadas

As funções friend declaradas em uma classe aninhada são consideradas como pertencentes ao escopo da classe aninhada, não à classe delimitadora. Portanto, as funções friend não ganham privilégios de acesso especiais aos membros ou às funções de membro da classe delimitadora. Se você quiser usar um nome que seja declarado em uma classe aninhada em uma função friend e essa função estiver definida no escopo do arquivo, use nomes de tipos qualificados da seguinte maneira:

```

// friend_functions_and_nested_classes.cpp

#include <string.h>

enum
{
    sizeOfMessage = 255
};

char *rgszMessage[sizeOfMessage];

class BufferedIO
{
public:
    class BufferedInput
    {
public:
        friend int GetExtendedErrorStatus();
        static char *message;
        static int messageSize;
        int iMsgNo;
    };
};

char *BufferedIO::BufferedInput::message;
int BufferedIO::BufferedInput::messageSize;

int GetExtendedErrorStatus()
{
    int iMsgNo = 1; // assign arbitrary value as message number

    strcpy_s( BufferedIO::BufferedInput::message,
              BufferedIO::BufferedInput::messageSize,
              rgszMessage[ iMsgNo ] );

    return iMsgNo;
}

int main()
{
}

```

O código a seguir mostra a função `GetExtendedErrorStatus` declarada como uma função friend. Na função, que é definida no escopo do arquivo, uma mensagem é copiada de uma matriz estática para um membro da classe. Uma implementação melhor de `GetExtendedErrorStatus` é declará-lo como:

```
int GetExtendedErrorStatus( char *message )
```

Com a interface anterior, várias classes podem usar os serviços dessa função transmitindo um local de memória para o qual querem que a mensagem de erro seja copiada.

Confira também

[Classes e structs](#)

Tipos de classe anônima

02/09/2020 • 2 minutes to read • [Edit Online](#)

As classes podem ser anônimas, ou seja, elas podem ser declaradas sem um *identificador*. Isso é útil quando você substitui um nome de classe por um `typedef` nome, como no seguinte:

```
typedef struct
{
    unsigned x;
    unsigned y;
} POINT;
```

NOTE

O uso de classes anônimas mostradas no exemplo anterior é útil para preservar a compatibilidade com o código C existente. Em alguns códigos C, o uso de `typedef` em conjunto com estruturas anônimas é predominante.

As classes anônimas também são úteis quando você quer que uma referência a um membro da classe apareça como se não estivesse contida em uma classe separada, como a seguir:

```
struct PTValue
{
    POINT ptLoc;
    union
    {
        int iValue;
        long lValue;
    };
};

PTValue ptv;
```

No código anterior, `iValue` pode ser acessado usando o operador de seleção de membro de objeto (.) da seguinte maneira:

```
int i = ptv.iValue;
```

As classes anônimas estão sujeitas a determinadas restrições. (Para obter mais informações sobre uniões anônimas, consulte [uniões](#).) Classes anônimas:

- Não é possível ter um construtor ou um destruidor.
- Não pode ser passado como argumentos para funções (a menos que a verificação de tipo seja derrotada usando reticências).
- Não é possível ser retornado como valores de retorno de funções.

Structs anônimos

Específico da Microsoft

Uma extensão do Microsoft C permite que você declare uma variável de estrutura dentro de outra estrutura sem

nomeá-la. Essas estruturas aninhadas são chamadas de estruturas anônimas. O C++ não permite estruturas anônimas.

Você pode acessar os membros de uma estrutura anônima como se fossem membros da estrutura que os contém.

```
// anonymous_structures.c
#include <stdio.h>

struct phone
{
    int areacode;
    long number;
};

struct person
{
    char name[30];
    char gender;
    int age;
    int weight;
    struct phone; // Anonymous structure; no name needed
} Jim;

int main()
{
    Jim.number = 1234567;
    printf_s("%d\n", Jim.number);
}
//Output: 1234567
```

FINAL específico da Microsoft

Ponteiros para membros

02/09/2020 • 7 minutes to read • [Edit Online](#)

As declarações dos ponteiros para os membros são casos especiais de declarações do ponteiro. Eles são declarados usando a seguinte sequência:

os especificadores de classe de armazenamento aceitam *CV-qualificadores*, opte pelo especificador *MS-modificador*_{opt}. *Name CV-qualificadores* opt `::*` *cv-qualifiers*_{opt} *ID PM-inicializador*_{opt} `opt**` ; `**`

1. O especificador de declaração:

- Um especificador de classe de armazenamento opcional.
- Opcional `const` e `volatile` especificadores.
- O especificador de tipo: o nome de um tipo. É o tipo do membro a ser apontado, não a classe.

2. O declarador:

- Um modificador opcional específico da Microsoft. Para obter mais informações, consulte [modificadores específicos da Microsoft](#).
- O nome qualificado da classe que contém os membros a serem apontados.
- O `::` operador.
- O `*` operador.
- Opcional `const` e `volatile` especificadores.
- O identificador que nomeia o ponteiro para o membro.

3. Um inicializador opcional de ponteiro para membro:

- O `=` operador.
- O `&` operador.
- O nome qualificado da classe.
- O `::` operador.
- O nome de um membro não estático da classe do tipo apropriado.

Como sempre, vários declaradores (e quaisquer inicializadores associados) são permitidos em uma única declaração. Um ponteiro para membro pode não apontar para um membro estático da classe, um membro do tipo de referência ou `void`.

Um ponteiro para um membro de uma classe difere de um ponteiro normal: ele tem as duas informações de tipo para o tipo do membro e para a classe à qual o membro pertence. Um ponteiro normal (tem o endereço de) identifica somente um único objeto na memória. Um ponteiro para um membro de uma classe identifica esse membro em qualquer instância da classe. O exemplo a seguir declara uma classe, `Window`, e alguns ponteiros para os dados de membro.

```

// pointers_to_members1.cpp
class Window
{
public:
    Window();                                // Default constructor.
    Window( int x1, int y1,                  // Constructor specifying
            int x2, int y2 );                // Window size.
    bool SetCaption( const char *szTitle ); // Set window caption.
    const char *GetCaption();                // Get window caption.
    char *szWinCaption;                     // Window caption.
};

// Declare a pointer to the data member szWinCaption.
char * Window::* pwCaption = &Window::szWinCaption;
int main()
{
}

```

No exemplo anterior, `pwCaption` é um ponteiro para qualquer membro da classe `Window` que é do tipo `char*`. O tipo de `pwCaption` é `char * Window::*`. O fragmento de código a seguir declara ponteiros para as funções de membro `SetCaption` e `GetCaption`.

```

const char * (Window::* pfNWGC)() = &Window::GetCaption;
bool (Window::* pfNWSC)( const char * ) = &Window::SetCaption;

```

Os ponteiros `pfNWGC` e `pfNWSC` apontam para `GetCaption` e `SetCaption` da classe `Window`, respectivamente. O código copia informações para a legenda da janela diretamente usando o ponteiro para o membro `pwCaption`:

```

Window wMainWindow;
Window *pwChildWindow = new Window;
char *szUntitled = "Untitled - ";
int cUntitledLen = strlen( szUntitled );

strcpy_s( wMainWindow.*pwCaption, cUntitledLen, szUntitled );
(wMainWindow.*pwCaption)[cUntitledLen - 1] = '1'; // same as
// wMainWindow.SzWinCaption [cUntitledLen - 1] = '1';
strcpy_s( pwChildWindow->*pwCaption, cUntitledLen, szUntitled );
(pwChildWindow->*pwCaption)[cUntitledLen - 1] = '2'; // same as
// pwChildWindow->szWinCaption[cUntitledLen - 1] = '2';

```

A diferença entre os `.*` e `->*` operadores e (os operadores de ponteiro para membro) é que o `.*` operador seleciona os membros que receberam uma referência de objeto ou objeto, enquanto o `->*` operador seleciona Membros por meio de um ponteiro. Para obter mais informações sobre esses operadores, consulte [expressões com operadores de ponteiro para membro](#).

O resultado dos operadores de ponteiro para membro é o tipo do membro. Nesse caso, use `char *`.

O fragmento de código a seguir invoca as funções do membro `GetCaption` e `SetCaption` usando ponteiros para os membros:

```
// Allocate a buffer.
enum {
    sizeOfBuffer = 100
};
char szCaptionBase[sizeOfBuffer];

// Copy the main window caption into the buffer
// and append "[View 1]".
strcpy_s( szCaptionBase, sizeOfBuffer, (wMainWindow.*pfnwGC)() );
strcat_s( szCaptionBase, sizeOfBuffer, " [View 1]" );
// Set the child window's caption.
(pwChildWindow->*pfnwSC)( szCaptionBase );
```

Restrições em ponteiros para membros

O endereço de um membro estático não é um ponteiro para um membro. É um ponteiro regular para uma instância do membro estático. Existe apenas uma instância de um membro estático para todos os objetos de uma determinada classe. Isso significa que você pode usar os operadores de endereço (`&`) e de desreferência (`()`) comuns `*`.

Ponteiros para membros e funções virtuais

Invocar uma função virtual por meio de uma função de ponteiro para membro funciona como se a função tivesse sido chamada diretamente. A função correta é pesquisada na tabela `v` e invocada.

A chave para as funções virtuais funcionarem, como sempre, é chamá-las por meio de um ponteiro para uma classe base. (Para obter mais informações sobre as funções virtuais, consulte [funções virtuais](#).)

O código a seguir mostra como chamar uma função virtual com uma função de ponteiro para membro:

```
// virtual_functions.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

class Base
{
public:
    virtual void Print();
};

void (Base::* bfnPrint)() = &Base::Print;
void Base::Print()
{
    cout << "Print function for class Base" << endl;
}

class Derived : public Base
{
public:
    void Print(); // Print is still a virtual function.
};

void Derived::Print()
{
    cout << "Print function for class Derived" << endl;
}

int main()
{
    Base    *bPtr;
    Base    bObject;
    Derived dObject;
    bPtr = &bObject; // Set pointer to address of bObject.
    (bPtr->*bfnPrint)();
    bPtr = &dObject; // Set pointer to address of dObject.
    (bPtr->*bfnPrint)();
}

// Output:
// Print function for class Base
// Print function for class Derived
```

Ponteiro ::::no-loc(this):::

02/09/2020 • 6 minutes to read • [Edit Online](#)

O ::::no-loc(this)::: ponteiro é um ponteiro acessível somente dentro das funções de membro não estático de ::::no-loc(class)::: um ::::no-loc(struct)::: tipo, ou ::::no-loc(union):::. Ele aponta para o objeto para o qual a função de membro é chamada. Funções membro static não têm um ::::no-loc(this)::: ponteiro.

Sintaxe

```
::::no-loc(this):::  
::::no-loc(this):::->member-identifier
```

Comentários

O ponteiro de um objeto ::::no-loc(this)::: não faz parte do próprio objeto. Ele não é refletido no resultado de uma ::::no-loc(sizeof)::: instrução no objeto. Quando uma função de membro não estático é chamada para um objeto, o compilador passa o endereço do objeto para a função como um argumento oculto. Por exemplo, a chamada de função a seguir:

```
myDate.setMonth( 3 );
```

pode ser interpretado como:

```
setMonth( &myDate, 3 );
```

O endereço do objeto está disponível de dentro da função membro como o ::::no-loc(this)::: ponteiro. A maioria dos ::::no-loc(this)::: usos de ponteiro é implícita. É legal, embora desnecessário, usar um explícito ::::no-loc(this)::: ao fazer referência a membros do ::::no-loc(class):::. Por exemplo:

```
void Date::setMonth( int mn )  
{  
    month = mn;           // These three statements  
    ::::no-loc(this):::->month = mn;      // are equivalent  
    (*:::no-loc(this):::).month = mn;  
}
```

A expressão *:::no-loc(this)::: é normalmente usada para retornar o objeto atual de uma função de membro:

```
return *:::no-loc(this):::;
```

O ::::no-loc(this)::: ponteiro também é usado para proteger contra autoreferência:

```
if (&Object != ::::no-loc(this)::: ) {  
    // do not execute in cases of self-reference
```

NOTE

Como o `:::no-loc(this):::` ponteiro não pode ser modificado, as atribuições ao `:::no-loc(this):::` ponteiro não são permitidas. As implementações anteriores do C++ permitiram a atribuição `:::no-loc(this):::`.

Ocasionalmente, o `:::no-loc(this):::` ponteiro é usado diretamente — por exemplo, para manipular Ures de dados autoreferenciais `::no-loc(struct):::`, onde o endereço do objeto atual é necessário.

Exemplo

```

// ::::no-loc(this):::_pointer.cpp
// compile with: /EHsc

#include <iostream>
#include <string.h>

using namespace std;

::::no-loc(class)::: Buf
{
public:
    Buf( char* szBuffer, size_t sizeOfBuffer );
    Buf& operator=( ::::no-loc(const)::: Buf & );
    void Display() { cout << buffer << endl; }

private:
    char*    buffer;
    size_t   sizeOfBuffer;
};

Buf::Buf( char* szBuffer, size_t sizeOfBuffer )
{
    sizeOfBuffer++; // account for a NULL terminator

    buffer = new char[ sizeOfBuffer ];
    if (buffer)
    {
        strcpy_s( buffer, sizeOfBuffer, szBuffer );
        sizeOfBuffer = sizeOfBuffer;
    }
}

Buf& Buf::operator=( ::::no-loc(const)::: Buf &otherbuf )
{
    if( &otherbuf != ::::no-loc(this)::: )
    {
        if (buffer)
            delete [] buffer;

        sizeOfBuffer = strlen( otherbuf.buffer ) + 1;
        buffer = new char[sizeOfBuffer];
        strcpy_s( buffer, sizeOfBuffer, otherbuf.buffer );
    }
    return *::::no-loc(this):::;
}

int main()
{
    Buf myBuf( "my buffer", 10 );
    Buf yourBuf( "your buffer", 12 );

    // Display 'my buffer'
    myBuf.Display();

    // assignment operator
    myBuf = yourBuf;

    // Display 'your buffer'
    myBuf.Display();
}

```

```

my buffer
your buffer

```

Tipo do `:::no-loc(this):::` ponteiro

O `:::no-loc(this):::` tipo do ponteiro pode ser modificado na declaração da função pelas `:::no-loc(const):::` `:::no-loc(volatile):::` palavras-chave e. Para declarar uma função que tenha qualquer um desses atributos, adicione as palavras-chave após a lista de argumentos da função.

Considere um exemplo:

```
// type_of_:::no-loc(this):::_pointer1.cpp
:::no-loc(class)::: Point
{
    unsigned X() :::no-loc(const):::;
};
int main()
{}
```

O código anterior declara uma função de membro, `X`, na qual o `:::no-loc(this):::` ponteiro é tratado como um `:::no-loc(const):::` ponteiro para um `:::no-loc(const):::` objeto. As combinações de opções *CV-mod-List* podem ser usadas, mas sempre modificam o objeto apontado pelo `:::no-loc(this):::` ponteiro, não pelo próprio ponteiro. A declaração a seguir declara a função `X`, onde o `:::no-loc(this):::` ponteiro é um `:::no-loc(const):::` ponteiro para um `:::no-loc(const):::` objeto:

```
// type_of_:::no-loc(this):::_pointer2.cpp
:::no-loc(class)::: Point
{
    unsigned X() :::no-loc(const):::;
};
int main()
{}
```

O tipo de `:::no-loc(this):::` em uma função de membro é descrito pela sintaxe a seguir. A *lista CV-Qualifier*-é determinada do Declarador da função membro. Ele pode ser `:::no-loc(const):::` ou `:::no-loc(volatile):::` (ou ambos). * `:::no-loc(class)::: -Type*` é o nome do `:::no-loc(class):::`:

`[CV-Qualifier-List] * :::no-loc(class)::: -tipo* *** :::no-loc(const)::: :::no-loc(this)::: **`

Em outras palavras, o `:::no-loc(this):::` ponteiro é sempre um `:::no-loc(const):::` ponteiro. Ele não pode ser reatribuído. Os `:::no-loc(const):::` `:::no-loc(volatile):::` qualificadores ou usados na declaração da função membro se aplicam à `:::no-loc(class):::` instância `:::no-loc(this):::` apontada pelo ponteiro em, no escopo dessa função.

A tabela a seguir explica mais sobre como esses modificadores funcionam.

Semântica de `:::no-loc(this):::` modificadores

MODIFICADOR	SIGNIFICADO
<code>:::no-loc(const):::</code>	Não é possível alterar os dados do membro; Não é possível invocar funções de membro que não são <code>:::no-loc(const):::</code> .
<code>:::no-loc(volatile):::</code>	Os dados do membro são carregados da memória toda vez que são acessados; desabilita determinadas otimizações.

É um erro passar um `:::no-loc(const):::` objeto para uma função membro que não é `:::no-loc(const):::`.

Da mesma forma, também é um erro passar um `:::no-loc(volatile):::` objeto para uma função de membro que não é `:::no-loc(volatile):::`.

Funções de membro declaradas como `:::no-loc(const):::` não podem alterar dados de membro — nessas funções, o `:::no-loc(this):::` ponteiro é um ponteiro para um `:::no-loc(const):::` objeto.

NOTE

Con `:::no-loc(struct):::` ORS e de `:::no-loc(struct):::` ORS não podem ser declarados como `:::no-loc(const):::` ou `:::no-loc(volatile):::`. No entanto, eles podem ser invocados em `:::no-loc(const):::` `:::no-loc(volatile):::` objetos ou.

Confira também

[Palavras-chave](#)

Campos de bit C++

02/09/2020 • 4 minutes to read • [Edit Online](#)

As classes e as estruturas podem conter membros que ocupam menos armazenamento do que um tipo integral. Esses membros são especificados como campos de bits. A sintaxe da especificação de *Declarador de membro de campo de bits* segue:

Sintaxe

Declarador: expressão de constante

Comentários

O *Declarador* (opcional) é o nome pelo qual o membro é acessado no programa. Deve ser um tipo integral (incluindo tipos enumerados). A expressão *Constant* especifica o número de bits que o membro ocupa na estrutura. Campos de bits anônimos - ou seja, membros de campos de bits sem identificador - podem ser usados para preenchimento.

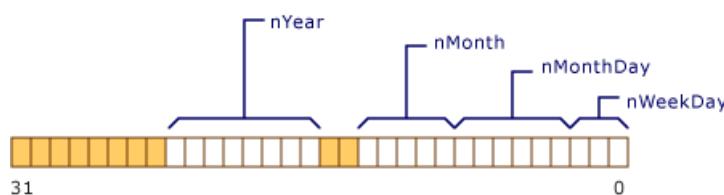
NOTE

Um campo de bits não nomeado de largura 0 força o alinhamento do próximo campo de bits para o próximo limite de **tipo**, em que **Type** é o tipo do membro.

O exemplo a seguir declara uma estrutura que contém campos de bits:

```
// bit_fields1.cpp
// compile with: /LD
struct Date {
    unsigned short nWeekDay : 3;      // 0..7  (3 bits)
    unsigned short nMonthDay : 6;      // 0..31 (6 bits)
    unsigned short nMonth : 5;        // 0..12 (5 bits)
    unsigned short nYear : 8;         // 0..100 (8 bits)
};
```

O layout de memória conceitual de um objeto do tipo `Date` é mostrado na figura a seguir.



Layout de memória de um objeto Date

Observe que `nYear` tem 8 bits de comprimento e excederia o limite de palavras do tipo declarado, `unsigned short`. Portanto, ele é iniciado no início de um novo `unsigned short`. Não é necessário que todos os campos de bits caibam em um mesmo objeto do tipo subjacente; novas unidades de armazenamento são alocadas de acordo com o número de bits solicitados na declaração.

Específico da Microsoft

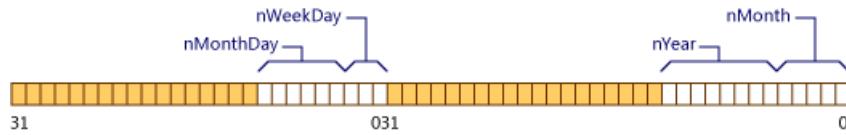
A ordenação dos dados declarados como campos de bits vai do bit inferior até o superior, como mostra a figura acima.

FINAL específico da Microsoft

Se a declaração de uma estrutura inclui um campo não nomeado de comprimento 0, como mostrado no exemplo a seguir,

```
// bit_fields2.cpp
// compile with: /LD
struct Date {
    unsigned nWeekDay : 3;    // 0..7  (3 bits)
    unsigned nMonthDay : 6;   // 0..31 (6 bits)
    unsigned : 0;           // Force alignment to next boundary.
    unsigned nMonth : 5;    // 0..12 (5 bits)
    unsigned nYear : 8;     // 0..100 (8 bits)
};
```

em seguida, o layout de memória é mostrado na figura a seguir:



Layout do objeto Date com campo de bits de comprimento zero

O tipo subjacente de um campo de bits deve ser um tipo integral, conforme descrito em [tipos internos](#).

Se o inicializador de uma referência do tipo `const T&` for um lvalue que se refere a um campo de bits do tipo `T`, a referência não será associada ao campo de bits diretamente. Em vez disso, a referência é associada a um inicializador temporário para conter o valor do campo de bits.

Restrições em campos de bits

A lista a seguir detalha as operações erradas em campos de bits:

- Obtendo o endereço de um campo de bits.
- Inicializando uma não `const` referência com um campo de bits.

Confira também

[Classes e structs](#)

Expressões lambda em C++

02/09/2020 • 20 minutes to read • [Edit Online](#)

No C++ 11 e posteriores, uma expressão lambda – geralmente chamada de *lambda* – é uma maneira conveniente de definir um objeto de função anônima (um *fechamento*) diretamente no local onde ele é invocado ou passado como um argumento para uma função. Normalmente, as Lambdas são usadas para encapsular algumas linhas de código que são passadas para algoritmos ou métodos assíncronos. Este artigo define o que são as lambdas, as compara a outras técnicas de programação, descreve suas vantagens e fornece um exemplo básico.

Tópicos Relacionados

- [Expressões lambda versus objetos de função](#)
- [Trabalhando com expressões lambda](#)
- [Expressões lambda constexpr](#)

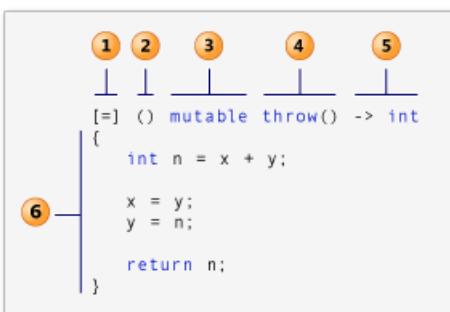
Partes de uma expressão lambda

O padrão ISO C++ mostra um lambda simples que é passado como o terceiro argumento para a `std::sort()` função:

```
#include <algorithm>
#include <cmath>

void abssort(float* x, unsigned n) {
    std::sort(x, x + n,
              // Lambda expression begins
              [] (float a, float b) {
                  return (std::abs(a) < std::abs(b));
              } // end of lambda expression
    );
}
```

Esta ilustração mostra as partes de um lambda:



1. *Capture a cláusula* (também conhecida como *lambda-apresentador* na especificação C++.)
2. *lista de parâmetros Adicional*. (Também conhecido como *Declarador lambda*)
3. *especificação mutável Adicional*.
4. *especificação de exceção Adicional*.
5. *tipo de retorno à direita Adicional*.

6. corpo lambda.

Cláusula capture

Um lambda pode introduzir novas variáveis em seu corpo (em C++ 14) e também pode acessar, ou *capturar*, variáveis do escopo ao redor. Um lambda começa com a cláusula Capture (*lambda-apresentador* na sintaxe padrão), que especifica quais variáveis são capturadas e se a captura é por valor ou por referência. Variáveis que têm o prefixo E comercial (&) são acessadas por referência e variáveis que não têm o prefixo são acessadas por valor.

Uma cláusula de captura vazia, [], indica que o corpo da expressão lambda não acessa variáveis no escopo delimitador.

Você pode usar o modo de captura padrão (*captura-padrão* na sintaxe padrão) para indicar como capturar todas as variáveis externas que são referenciadas no lambda: [&] significa que todas as variáveis que você faz referência são capturadas pelas referências e [=] significa que elas são capturadas por valor. Você pode usar um modo de captura padrão e especificar o modo oposto explicitamente para variáveis específicas. Por exemplo, se um corpo de lambda acessar a variável externa `total` por referência e a variável externa `factor` por valor, as seguintes cláusulas de captura serão equivalentes:

```
[&total, factor]
[factor, &total]
[&, factor]
[factor, &]
[=, &total]
[&total, =]
```

Somente variáveis mencionadas no lambda são capturadas quando um padrão de captura é usado.

Se uma cláusula Capture incluir uma captura-padrão &, não `identifier` em uma `capture` dessa cláusula de captura poderá ter o formulário & `identifier`. Da mesma forma, se a cláusula Capture incluir uma captura-padrão =, nenhuma `capture` dessa cláusula Capture poderá ter o formulário = `identifier`. Um identificador ou `this` não pode aparecer mais de uma vez em uma cláusula Capture. O snippet de código a seguir ilustra alguns exemplos.

```
struct S { void f(int i); };

void S::f(int i) {
    [&, i]{};      // OK
    [&, &i]{};     // ERROR: i preceded by & when & is the default
    [=, this]{};   // ERROR: this when = is the default
    [=, *this]{};  // OK: captures this by value. See below.
    [i, i]{};      // ERROR: i repeated
}
```

Uma captura seguida por uma elipse é uma expansão de pacote, conforme mostrado neste exemplo de [modelo Variadic](#):

```
template<class... Args>
void f(Args... args) {
    auto x = [args...] { return g(args...); };
    x();
}
```

Para usar expressões lambda no corpo de um método de classe, passe o `this` ponteiro para a cláusula Capture para fornecer acesso aos métodos e membros de dados da classe delimitadora.

Visual Studio 2017 versão 15,3 e posterior (disponível com [/std: c++ 17](#)): o `this` ponteiro pode ser capturado por valor especificando `*this` na cláusula Capture. Captura por valor significa que o *fechamento* inteiro, que é o objeto de função anônima que encapsula a expressão lambda, é copiado para cada site de chamada onde o lambda é invocado. A captura por valor é útil quando o lambda será executado em operações paralelas ou assíncronas, especialmente em determinadas arquiteturas de hardware, como o NUMA.

Para obter um exemplo que mostra como usar expressões lambda com métodos de classe, consulte "exemplo: usando uma expressão lambda em um método" em [exemplos de expressões lambda](#).

Ao usar a cláusula de captura, nós recomendamos que você mantenha esses pontos em mente, especialmente ao usar lambdas com multithreading:

- As capturas de referência podem ser usadas para modificar variáveis externas, mas as capturas de valor não. (`mutable` permite que as cópias sejam modificadas, mas não originais).
- As capturas de referência refletem atualizações para variáveis externas, mas as capturas de valor não.
- As capturas de referência introduzem uma dependência de tempo de vida, mas as capturas de valor não possuem dependências de tempo de vida. Isso é especialmente importante quando o lambda é executado de forma assíncrona. Se você capturar um local por referência em um lambda assíncrono, esse local possivelmente estará no momento em que o lambda é executado, resultando em uma violação de acesso no tempo de execução.

Captura generalizada (C++ 14)

No C++ 14, você pode introduzir e inicializar novas variáveis na cláusula Capture, sem a necessidade de ter essas variáveis no escopo delimitador da função lambda. A inicialização pode ser expressa como qualquer expressão arbitrária; o tipo da nova variável é deduzido do tipo produzido pela expressão. Um benefício desse recurso é que, no C++ 14, você pode capturar variáveis somente de movimentação (como `std:: unique_ptr`) do escopo ao redor e usá-las em um lambda.

```
pNums = make_unique<vector<int>>(nums);
//...
auto a = [ptr = move(pNums)]()
{
    // use ptr
};
```

Lista de parâmetros

Além de capturar variáveis, um lambda pode aceitar parâmetros de entrada. Uma lista de parâmetros (*Declarador lambda* na sintaxe padrão) é opcional e, na maioria dos aspectos, é semelhante à lista de parâmetros para uma função.

```
auto y = [] (int first, int second)
{
    return first + second;
};
```

No C++ 14, se o tipo de parâmetro for genérico, você poderá usar a `auto` palavra-chave como o especificador de tipo. Isso informa o compilador para criar o operador de chamada de função como um modelo. Cada instância do `auto` em uma lista de parâmetros é equivalente a um parâmetro de tipo distinto.

```
auto y = [] (auto first, auto second)
{
    return first + second;
};
```

Uma expressão lambda pode usar outra expressão lambda como seu argumento. Para obter mais informações, consulte "expressões lambda de ordem superior" no tópico [exemplos de expressões lambda](#).

Como uma lista de parâmetros é opcional, você pode omitir os parênteses vazios se não passar argumentos para a expressão lambda e seu Declarador lambda não contiver uma *especificação de exceção, de retorno à direita* ou `mutable`.

Especificação mutável

Normalmente, um operador de chamada de função de lambda é constante por valor, mas o uso da `mutable` palavra-chave cancela isso. Ele não produz membros de dados mutáveis. A especificação mutável permite que o corpo de uma expressão lambda modifique variáveis capturadas por valor. Alguns dos exemplos mais adiante neste artigo mostram como usar o `mutable`.

Especificação de exceção

Você pode usar a `noexcept` especificação de exceção para indicar que a expressão lambda não lança nenhuma exceção. Assim como acontece com funções comuns, o compilador do Microsoft C++ gera o aviso [C4297](#) se uma expressão lambda declara a `noexcept` especificação de exceção e o corpo lambda gera uma exceção, como mostrado aqui:

```
// throw_lambda_expression.cpp
// compile with: /W4 /EHsc
int main() // C4297 expected
{
    []() noexcept { throw 5; }();
}
```

Para obter mais informações, consulte [especificações de exceção \(throw\)](#).

Tipo de retorno

O tipo de retorno de uma expressão lambda é deduzido automaticamente. Você não precisa usar a `auto` palavra-chave, a menos que especifique um *tipo de retorno à direita*. O *tipo de retorno à direita* é semelhante à parte de tipo de retorno de um método ou função comum. No entanto, o tipo de retorno deve seguir a lista de parâmetros e você deve incluir a palavra-chave de tipo de retorno à direita `->` antes do tipo de retorno.

É possível omitir a parte return-type de uma expressão lambda se o corpo lambda contiver apenas uma instrução de retorno ou se a expressão lambda não retornar um valor. Se o corpo lambda contém uma instrução de retorno, o compilador deduzirá o tipo de retorno do tipo da expressão de retorno. Caso contrário, o compilador deduzirá o tipo de retorno como `void`. Considere os snippets de código do exemplo a seguir que ilustram esse princípio.

```
auto x1 = [](int i){ return i; }; // OK: return type is int
auto x2 = []{ return{ 1, 2 }; }; // ERROR: return type is void, deducing
                                // return type from braced-init-list is not valid
```

Uma expressão lambda pode gerar outra expressão lambda como seu valor de retorno. Para obter mais informações, consulte "expressões lambda de ordem superior" em [exemplos de expressões lambda](#).

Corpo lambda

O corpo lambda (*instrução composta* na sintaxe padrão) de uma expressão lambda pode conter qualquer coisa que o corpo de um método ou função comum possa conter. O corpo de uma função comum e de uma expressão lambda pode acessar os seguintes tipos de variáveis:

- Variáveis capturadas do escopo delimitador, conforme descrito anteriormente.
- parâmetros

- Variáveis declaradas localmente
- Membros de dados de classe, quando declarados dentro de uma classe e `this` são capturados
- Qualquer variável que possui a duração de armazenamento estático como, por exemplo, variáveis globais

O exemplo a seguir contém uma expressão lambda que captura explicitamente a variável `n` por valor e que captura implicitamente a variável `m` por referência:

```
// captures_lambda_expression.cpp
// compile with: /W4 /EHsc
#include <iostream>
using namespace std;

int main()
{
    int m = 0;
    int n = 0;
    [&, n] (int a) mutable { m = ++n + a; }(4);
    cout << m << endl << n << endl;
}
```

```
5
0
```

Como a variável `n` é capturada pelo valor, seu valor permanece `0` após a chamada para a expressão lambda. A `mutable` especificação permite que `n` seja modificada dentro do lambda.

Embora uma expressão lambda possa capturar apenas variáveis que tenham a duração automática de armazenamento, você pode usar variáveis que tenham a duração de armazenamento estático no corpo de uma expressão lambda. O exemplo a seguir usa a função `generate` e uma expressão lambda para atribuir um valor para cada elemento em um objeto `vector`. A expressão lambda modifica a variável estática para gerar o valor do próximo elemento.

```
void fillVector(vector<int>& v)
{
    // A local static variable.
    static int nextValue = 1;

    // The lambda expression that appears in the following call to
    // the generate function modifies and uses the local static
    // variable nextValue.
    generate(v.begin(), v.end(), [] { return nextValue++; });
    //WARNING: this is not thread-safe and is shown for illustration only
}
```

Para obter mais informações, consulte [gerar](#).

O exemplo de código a seguir usa a função do exemplo anterior e adiciona um exemplo de uma expressão lambda que usa o algoritmo de biblioteca padrão C++ `generate_n`. Essa expressão lambda atribui um elemento de um objeto `vector` à soma dos dois elementos anteriores. A `mutable` palavra-chave é usada para que o corpo da expressão lambda possa modificar suas cópias das variáveis externas `x` e `y`, que a expressão lambda captura por valor. Uma vez que a expressão lambda captura as variáveis originais `x` e `y` por valor, seus valores permanecem `1` depois que a lambda é executada.

```
// compile with: /W4 /EHsc
#include <algorithm>
```

```

----- --8-- -----
#include <iostream>
#include <vector>
#include <string>

using namespace std;

template <typename C> void print(const string& s, const C& c) {
    cout << s;

    for (const auto& e : c) {
        cout << e << " ";
    }

    cout << endl;
}

void fillVector(vector<int>& v)
{
    // A local static variable.
    static int nextValue = 1;

    // The lambda expression that appears in the following call to
    // the generate function modifies and uses the local static
    // variable nextValue.
    generate(v.begin(), v.end(), [] { return nextValue++; });
    //WARNING: this is not thread-safe and is shown for illustration only
}

int main()
{
    // The number of elements in the vector.
    const int elementCount = 9;

    // Create a vector object with each element set to 1.
    vector<int> v(elementCount, 1);

    // These variables hold the previous two elements of the vector.
    int x = 1;
    int y = 1;

    // Sets each element in the vector to the sum of the
    // previous two elements.
    generate_n(v.begin() + 2,
               elementCount - 2,
               [=]() mutable throw() -> int { // lambda is the 3rd parameter
                   // Generate current value.
                   int n = x + y;
                   // Update previous two values.
                   x = y;
                   y = n;
                   return n;
               });
    print("vector v after call to generate_n() with lambda: ", v);

    // Print the local variables x and y.
    // The values of x and y hold their initial values because
    // they are captured by value.
    cout << "x: " << x << " y: " << y << endl;

    // Fill the vector with a sequence of numbers
    fillVector(v);
    print("vector v after 1st call to fillVector(): ", v);
    // Fill the vector with the next sequence of numbers
    fillVector(v);
    print("vector v after 2nd call to fillVector(): ", v);
}

```

```
vector v after call to generate_n() with lambda: 1 1 2 3 5 8 13 21 34
x: 1 y: 1
vector v after 1st call to fillVector(): 1 2 3 4 5 6 7 8 9
vector v after 2nd call to fillVector(): 10 11 12 13 14 15 16 17 18
```

Para obter mais informações, consulte [generate_n](#).

constexpr expressões lambda

Visual Studio 2017 versão 15,3 e posterior (disponível com `/std:c++17`): uma expressão lambda pode ser declarada como `constexpr` ou usada em uma expressão constante quando a inicialização de cada membro de dados que captura ou apresenta é permitida em uma expressão constante.

```
int y = 32;
auto answer = [y]() constexpr
{
    int x = 10;
    return y + x;
};

constexpr int Increment(int n)
{
    return [n] { return n + 1; }();
```

Um lambda é implicitamente `constexpr` se o resultado satisfizer os requisitos de uma `constexpr` função:

```
auto answer = [](int n)
{
    return 32 + n;
};

constexpr int response = answer(10);
```

Se uma lambda for implícita ou explicitamente `constexpr`, a conversão para um ponteiro de função produzirá uma `constexpr` função:

```
auto Increment = [](int n)
{
    return n + 1;
};

constexpr int(*inc)(int) = Increment;
```

Específico da Microsoft

Não há suporte para lambdas nas seguintes entidades gerenciadas do Common Language Runtime (CLR):

`ref class` , `ref struct` , `value class` ou `value struct` .

Se você estiver usando um modificador específico da Microsoft `__declspec` , como, poderá inseri-lo em uma expressão lambda imediatamente após o `parameter-declaration-clause` – por exemplo:

```
auto Sqr = [](int t) __declspec(code_seg("PagedMem")) -> int { return t*t; };
```

Para determinar se um modificador é suportado por lambdas, consulte o artigo sobre ele na seção de

modificadores específicos da Microsoft da documentação.

Além da funcionalidade lambda padrão do C++ 11, o Visual Studio dá suporte a lambdas sem monitoração de estado, que são conversíveis pelo Omni para funções que usam convenções de chamada arbitrárias.

Confira também

[Referência da linguagem C++](#)

[Objetos de função na biblioteca padrão C++](#)

[Chamada de função](#)

[for_each](#)

Sintaxe da expressão lambda

25/03/2020 • 6 minutes to read • [Edit Online](#)

Esse artigo demonstra a sintaxe e os elementos estruturais das expressões lambda. Para obter uma descrição das expressões lambda, consulte [expressões lambda](#).

Objetos de função vs. lambdas

Quando você escreve o código, provavelmente usa ponteiros de função e objetos de função para resolver problemas e executar cálculos, especialmente quando você usa [C++ algoritmos de biblioteca padrão](#). Os ponteiros de função e os objetos de função têm vantagens e desvantagens — por exemplo, os ponteiros de função têm sobrecarga sintática mínima, mas não retêm o estado em um escopo, e os objetos de função podem manter o estado, mas exigem a sobrecarga sintática de um definição de classe.

Um lambda combina os benefícios dos ponteiros de função com os objetos de função e evita suas desvantagens. Como os objetos de função, um lambda é flexível e pode manter o estado, mas ao contrário de um objeto de função, sua sintaxe de compactação não requer uma definição de classe explícita. Ao usar lambdas, você pode escrever código que seja menos inconveniente e menos sujeito a erros do que o código para um objeto de função equivalente.

Os exemplos a seguir comparam o uso de uma lambda ao uso de um objeto de função. O primeiro exemplo usa uma lambda para imprimir no console se cada elemento em um objeto `vector` é par ou ímpar. O segundo exemplo usa um objeto de função para realizar a mesma tarefa.

Exemplo 1: Usando uma lambda

Este exemplo passa um lambda para a função `for_each`. O lambda imprime um resultado que indica se cada elemento em um objeto `vector` é par ou ímpar.

Código

```

// even_lambda.cpp
// compile with: cl /EHsc /nologo /W4 /MTd
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    // Create a vector object that contains 9 elements.
    vector<int> v;
    for (int i = 1; i < 10; ++i) {
        v.push_back(i);
    }

    // Count the number of even numbers in the vector by
    // using the for_each function and a lambda.
    int evenCount = 0;
    for_each(v.begin(), v.end(), [&evenCount] (int n) {
        cout << n;
        if (n % 2 == 0) {
            cout << " is even " << endl;
            ++evenCount;
        } else {
            cout << " is odd " << endl;
        }
    });

    // Print the count of even numbers to the console.
    cout << "There are " << evenCount
        << " even numbers in the vector." << endl;
}

```

```

1 is odd
2 is even
3 is odd
4 is even
5 is odd
6 is even
7 is odd
8 is even
9 is odd
There are 4 even numbers in the vector.

```

Comentários

No exemplo, o terceiro argumento para a função `for_each` é um lambda. A parte `[&evenCount]` especifica a cláusula capture da expressão, `(int n)` especifica a lista de parâmetros e a parte restante especifica o corpo da expressão.

Exemplo 2: Usando um objeto de função

Às vezes, uma lambda seria muito pesada para se estender muito além do exemplo anterior. O exemplo a seguir usa um objeto de função em vez de um lambda, junto com a função `for_each`, para produzir os mesmos resultados que o exemplo 1. Os dois exemplos armazenam a contagem de números pares em um objeto `vector`. Para manter o estado da operação, a classe `FunctorClass` armazena a variável `m_evenCount` por referência como uma variável de membro. Para executar a operação, `FunctorClass` implementa o operador de chamada de função, `operador ()`. O compilador C++ da Microsoft gera código que é comparável em tamanho e desempenho para o código lambda no exemplo 1. Para um problema básico como o deste artigo, o design da lambda mais simples é provavelmente melhor do que o design do objeto de função. No entanto, se achar que a funcionalidade pode exigir expansão significativa no futuro, use um design de objeto de função, assim, a manutenção do código será

mais fácil.

Para obter mais informações sobre o **operador ()** , consulte [chamada de função](#). Para obter mais informações sobre a função **for_each** , consulte [for_each](#).

Código

```
// even_functor.cpp
// compile with: /EHsc
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

class FunctorClass
{
public:
    // The required constructor for this example.
    explicit FunctorClass(int& evenCount)
        : m_evenCount(evenCount) { }

    // The function-call operator prints whether the number is
    // even or odd. If the number is even, this method updates
    // the counter.
    void operator()(int n) const {
        cout << n;

        if (n % 2 == 0) {
            cout << " is even " << endl;
            ++m_evenCount;
        } else {
            cout << " is odd " << endl;
        }
    }
}

private:
    // Default assignment operator to silence warning C4512.
    FunctorClass& operator=(const FunctorClass&);

    int& m_evenCount; // the number of even variables in the vector.
};

int main()
{
    // Create a vector object that contains 9 elements.
    vector<int> v;
    for (int i = 1; i < 10; ++i) {
        v.push_back(i);
    }

    // Count the number of even numbers in the vector by
    // using the for_each function and a function object.
    int evenCount = 0;
    for_each(v.begin(), v.end(), FunctorClass(evenCount));

    // Print the count of even numbers to the console.
    cout << "There are " << evenCount
        << " even numbers in the vector." << endl;
}
```

```
1 is odd
2 is even
3 is odd
4 is even
5 is odd
6 is even
7 is odd
8 is even
9 is odd
There are 4 even numbers in the vector.
```

Confira também

[Expressões Lambda](#)

[Exemplos de expressões lambda](#)

[generate](#)

[generate_n](#)

[for_each](#)

[Especificações de exceção \(lançar\)](#)

[Aviso do compilador \(nível 1\) C4297](#)

[Modificadores específicos da Microsoft](#)

Exemplos de expressões lambda

02/09/2020 • 14 minutes to read • [Edit Online](#)

Este artigo mostra como usar expressões lambda em seus programas. Para obter uma visão geral das expressões lambda, consulte [expressões lambda](#). Para obter mais informações sobre a estrutura de uma expressão lambda, consulte [sintaxe de expressão lambda](#).

Declarando expressões lambda

Exemplo 1

Como uma expressão lambda é digitada, você pode atribuí-la a uma `auto` variável ou a um `function` objeto, como mostrado aqui:

Código

```
// declaring_lambda_expressions1.cpp
// compile with: /EHsc /W4
#include <functional>
#include <iostream>

int main()
{
    using namespace std;

    // Assign the lambda expression that adds two numbers to an auto variable.
    auto f1 = [] (int x, int y) { return x + y; };

    cout << f1(2, 3) << endl;

    // Assign the same lambda expression to a function object.
    function<int(int, int)> f2 = [] (int x, int y) { return x + y; };

    cout << f2(3, 4) << endl;
}
```

Saída

```
5
7
```

Comentários

Para obter mais informações, consulte [auto](#), [function](#) classe e chamada de função.

Embora as expressões lambda sejam geralmente declaradas no corpo de uma função, você pode declará-las em qualquer lugar que possa inicializar uma variável.

Exemplo 2

O compilador do Microsoft C++ associa uma expressão lambda a suas variáveis capturadas quando a expressão é declarada em vez de quando a expressão é chamada. O exemplo a seguir mostra uma expressão lambda que captura a variável local `i` por valor e a variável local `j` por referência. Como a expressão lambda é capturada `i` por valor, a reatribuição posterior de `i` no programa não afetará o resultado da expressão. No entanto, como a expressão lambda captura `j` por referência, a reatribuição de `j` afetará o resultado da expressão.

Código

```
// declaring_lambda_expressions2.cpp
// compile with: /EHsc /W4
#include <functional>
#include <iostream>

int main()
{
    using namespace std;

    int i = 3;
    int j = 5;

    // The following lambda expression captures i by value and
    // j by reference.
    function<int (void)> f = [i, &j] { return i + j; };

    // Change the values of i and j.
    i = 22;
    j = 44;

    // Call f and print its result.
    cout << f() << endl;
}
```

Saída

```
47
```

[Neste [artigo](#)]

Chamando expressões lambda

Você poderá chamar uma expressão lambda imediatamente, conforme mostrado no próximo snippet de código. O segundo trecho mostra como passar um lambda como um argumento para algoritmos de biblioteca padrão do C++, como `find_if`.

Exemplo 1

Este exemplo declara uma expressão lambda que retorna a soma de dois inteiros e chama a expressão imediatamente com os argumentos `5` e `4`:

Código

```
// calling_lambda_expressions1.cpp
// compile with: /EHsc
#include <iostream>

int main()
{
    using namespace std;
    int n = [] (int x, int y) { return x + y; }(5, 4);
    cout << n << endl;
}
```

Saída

```
9
```

Exemplo 2

Este exemplo passa uma expressão lambda como argumento para a função `find_if`. A expressão lambda retorna `true` se seu parâmetro for um número par.

Código

```
// calling_lambda_expressions2.cpp
// compile with: /EHsc /W4
#include <list>
#include <algorithm>
#include <iostream>

int main()
{
    using namespace std;

    // Create a list of integers with a few initial elements.
    list<int> numbers;
    numbers.push_back(13);
    numbers.push_back(17);
    numbers.push_back(42);
    numbers.push_back(46);
    numbers.push_back(99);

    // Use the find_if function and a lambda expression to find the
    // first even number in the list.
    const list<int>::const_iterator result =
        find_if(numbers.begin(), numbers.end(),[](int n) { return (n % 2) == 0; });

    // Print the result.
    if (result != numbers.end()) {
        cout << "The first even number in the list is " << *result << endl;
    } else {
        cout << "The list contains no even numbers." << endl;
    }
}
```

Saída

```
The first even number in the list is 42.
```

Comentários

Para obter mais informações sobre a `find_if` função, consulte [find_if](#). Para obter mais informações sobre as funções de biblioteca padrão do C++ que executam algoritmos comuns, consulte [<algorithm>](#).

[Neste [artigo](#)]

Aninhando expressões lambda

Exemplo

É possível aninhar uma expressão lambda dentro de outra, como mostrado neste exemplo. A expressão lambda interna multiplica seu argumento por 2 e retorna o resultado. A expressão lambda externa chama a expressão lambda interna com seu argumento e adiciona 3 ao resultado.

Código

```
// nesting_lambda_expressions.cpp
// compile with: /EHsc /W4
#include <iostream>

int main()
{
    using namespace std;

    // The following lambda expression contains a nested lambda
    // expression.
    int timestwoplusthree = [](int x) { return [](int y) { return y * 2; }(x) + 3; }(5);

    // Print the result.
    cout << timestwoplusthree << endl;
}
```

Saída

```
13
```

Comentários

Neste exemplo, `[](int y) { return y * 2; }` representa a expressão lambda aninhada.

[\[Neste artigo\]](#)

Funções lambda de ordem superior

Exemplo

Muitas linguagens de programação dão suporte ao conceito de uma *função de ordem superior*. Uma função de ordem superior é uma expressão lambda que usa outra expressão lambda como seu argumento ou retorna uma expressão lambda. Você pode usar a `function` classe para habilitar uma expressão lambda C++ para se comportar como uma função de ordem superior. O exemplo a seguir mostra uma expressão lambda que retorna um objeto `function` e uma expressão lambda que usa um objeto `function` como seu argumento.

Código

```

// higher_order_lambda_expression.cpp
// compile with: /EHsc /W4
#include <iostream>
#include <functional>

int main()
{
    using namespace std;

    // The following code declares a lambda expression that returns
    // another lambda expression that adds two numbers.
    // The returned lambda expression captures parameter x by value.
    auto addtwointegers = [](int x) -> function<int(int)> {
        return [=](int y) { return x + y; };
    };

    // The following code declares a lambda expression that takes another
    // lambda expression as its argument.
    // The lambda expression applies the argument z to the function f
    // and multiplies by 2.
    auto higherorder = [](const function<int(int)>& f, int z) {
        return f(z) * 2;
    };

    // Call the lambda expression that is bound to higherorder.
    auto answer = higherorder(addtwointegers(7), 8);

    // Print the result, which is (7+8)*2.
    cout << answer << endl;
}

```

Saída

30

[\[Neste artigo\]](#)

Usando uma expressão lambda em uma função

Exemplo

Você pode usar expressões lambda no corpo de uma função. A expressão lambda pode acessar qualquer função ou membro de dados que a função delimitadora possa acessar. Você pode capturar o ponteiro de forma explícita ou implícita `this` para fornecer acesso a funções e membros de dados da classe delimitadora. **Visual Studio 2017 versão 15,3 e posterior** (disponível com `/std:c++17`): captura `this` por valor (`[*this]`) quando o lambda será usado em operações assíncronas ou paralelas em que o código pode ser executado depois que o objeto original sair do escopo.

Você pode usar o `this` ponteiro explicitamente em uma função, como mostrado aqui:

```

// capture "this" by reference
void ApplyScale(const vector<int>& v) const
{
    for_each(v.begin(), v.end(),
        [this](int n) { cout << n * _scale << endl; });
}

// capture "this" by value (Visual Studio 2017 version 15.3 and later)
void ApplyScale2(const vector<int>& v) const
{
    for_each(v.begin(), v.end(),
        [*this](int n) { cout << n * _scale << endl; });
}

```

Você também pode capturar o `this` ponteiro implicitamente:

```

void ApplyScale(const vector<int>& v) const
{
    for_each(v.begin(), v.end(),
        [=](int n) { cout << n * _scale << endl; });
}

```

O exemplo a seguir mostra a classe `Scale`, que encapsula um valor da escala.

```

// function_lambda_expression.cpp
// compile with: /EHsc /W4
#include <algorithm>
#include <iostream>
#include <vector>

using namespace std;

class Scale
{
public:
    // The constructor.
    explicit Scale(int scale) : _scale(scale) {}

    // Prints the product of each element in a vector object
    // and the scale value to the console.
    void ApplyScale(const vector<int>& v) const
    {
        for_each(v.begin(), v.end(), [=](int n) { cout << n * _scale << endl; });
    }

private:
    int _scale;
};

int main()
{
    vector<int> values;
    values.push_back(1);
    values.push_back(2);
    values.push_back(3);
    values.push_back(4);

    // Create a Scale object that scales elements by 3 and apply
    // it to the vector object. Does not modify the vector.
    Scale s(3);
    s.ApplyScale(values);
}

```

Saída

```
3
6
9
12
```

Comentários

A `ApplyScale` função usa uma expressão lambda para imprimir o produto do valor de escala e cada elemento em um `vector` objeto. A expressão lambda captura implicitamente `this` para que possa acessar o `_scale` membro.

[Neste artigo]

Usando expressões lambda com modelos

Exemplo

Como as expressões lambda são digitadas, é possível usá-las com modelos C++. O exemplo a seguir mostra as funções `negate_all` e `print_all`. A `negate_all` função aplica o unário `operator-` a cada elemento no `vector` objeto. A função `print_all` imprime cada elemento no objeto `vector` para o console.

Código

```
// template_lambda_expression.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <iostream>

using namespace std;

// Negates each element in the vector object. Assumes signed data type.
template <typename T>
void negate_all(vector<T>& v)
{
    for_each(v.begin(), v.end(), [](T& n) { n = -n; });
}

// Prints to the console each element in the vector object.
template <typename T>
void print_all(const vector<T>& v)
{
    for_each(v.begin(), v.end(), [](const T& n) { cout << n << endl; });
}

int main()
{
    // Create a vector of signed integers with a few elements.
    vector<int> v;
    v.push_back(34);
    v.push_back(-43);
    v.push_back(56);

    print_all(v);
    negate_all(v);
    cout << "After negate_all(): " << endl;
    print_all(v);
}
```

Saída

```
34
-43
56
After negate_all():
-34
43
-56
```

Comentários

Para obter mais informações sobre modelos C++, consulte [modelos](#).

[Neste artigo]

Manipulando exceções

Exemplo

O corpo de uma expressão lambda segue as regras para a manipulação de exceção estruturada (SEH) e o tratamento de exceções C++. Você pode manipular uma exceção gerada no corpo de uma expressão lambda ou adiar o tratamento de exceções para o escopo delimitador. O exemplo a seguir usa a `for_each` função e uma expressão lambda para preencher um `vector` objeto com os valores de outro. Ele usa um `try` / `catch` bloco para tratar o acesso inválido ao primeiro vetor.

Código

```
// eh_lambda_expression.cpp
// compile with: /EHsc /W4
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;

int main()
{
    // Create a vector that contains 3 elements.
    vector<int> elements(3);

    // Create another vector that contains index values.
    vector<int> indices(3);
    indices[0] = 0;
    indices[1] = -1; // This is not a valid subscript. It will trigger an exception.
    indices[2] = 2;

    // Use the values from the vector of index values to
    // fill the elements vector. This example uses a
    // try/catch block to handle invalid access to the
    // elements vector.
    try
    {
        for_each(indices.begin(), indices.end(), [&](int index) {
            elements.at(index) = index;
        });
    }
    catch (const out_of_range& e)
    {
        cerr << "Caught '" << e.what() << "'." << endl;
    };
}
```

Saída

```
Caught 'invalid vector<T> subscript'.
```

Comentários

Para obter mais informações sobre manipulação de exceção, consulte [tratamento de exceção](#).

[Neste artigo]

Usando expressões lambda com tipos gerenciados (C++/CLI)

Exemplo

A cláusula de captura de uma expressão lambda não pode conter uma variável que tenha um tipo gerenciado. No entanto, você pode passar um argumento que tenha um tipo gerenciado para uma lista de parâmetros de uma expressão lambda. O exemplo a seguir contém uma expressão lambda que captura a variável local não gerenciada `ch` por valor e usa um objeto `System.String` como seu parâmetro.

Código

```
// managed_lambda_expression.cpp
// compile with: /clr
using namespace System;

int main()
{
    char ch = '!'; // a local unmanaged variable

    // The following lambda expression captures local variables
    // by value and takes a managed String object as its parameter.
    [=](String ^s) {
        Console::WriteLine(s + Convert::ToChar(ch));
    }("Hello");
}
```

Saída

```
Hello!
```

Comentários

Você também pode usar expressões lambda com a biblioteca STL/CLR. Para obter mais informações, consulte [referência da biblioteca STL/CLR](#).

IMPORTANT

Não há suporte para lambdas nestas entidades gerenciadas do Common Language Runtime (CLR): `ref class` , `ref struct` , `value class` e `value struct` .

[Neste artigo]

Confira também

[Expressões lambda](#)

[Sintaxe de expressão lambda](#)

`auto`

`function` [Classes](#)

`find_if`

<algorithm>

[Chamada de função](#)

[Modelo](#)

[Tratamento de exceção](#)

[Referência da biblioteca STL/CLR](#)

expressões lambda constexpr em C++

02/09/2020 • 2 minutes to read • [Edit Online](#)

Visual Studio 2017 versão 15,3 e posterior (disponível com `/std: c++ 17`): uma expressão lambda pode ser declarada como `constexpr` ou usada em uma expressão constante quando a inicialização de cada membro de dados que captura ou apresenta é permitida em uma expressão constante.

```
int y = 32;
auto answer = [y]() constexpr
{
    int x = 10;
    return y + x;
};

constexpr int Increment(int n)
{
    return [n] { return n + 1; }();
}
```

Um lambda é implicitamente `constexpr` se o resultado satisfizer os requisitos de uma `constexpr` função:

```
auto answer = [](int n)
{
    return 32 + n;
};

constexpr int response = answer(10);
```

Se um lambda for implicitamente ou explicitamente `constexpr` e você o converter em um ponteiro de função, a função resultante também será `constexpr`:

```
auto Increment = [](int n)
{
    return n + 1;
};

constexpr int(*inc)(int) = Increment;
```

Confira também

[Referência da linguagem C++](#)

[Objetos de função na biblioteca padrão C++](#)

[Chamada de função](#)

[for_each](#)

Matrizes (C++)

02/09/2020 • 18 minutes to read • [Edit Online](#)

Uma matriz é uma sequência de objetos do mesmo tipo que ocupam uma área contígua de memória. As matrizes tradicionais de estilo C são a fonte de muitos bugs, mas ainda são comuns, especialmente em bases de código mais antigas. No C++ moderno, é altamente recomendável usar `std::vector` ou `std::array` em vez de matrizes de estilo C descritos nesta seção. Ambos os tipos de biblioteca padrão armazenam seus elementos como um bloco contíguo de memória. No entanto, eles fornecem segurança de tipo muito maior e iteradores de suporte que têm garantia de apontar para um local válido dentro da sequência. Para obter mais informações, consulte [contêineres \(C++ moderno\)](#).

Declarações de pilha

Em uma declaração de matriz do C++, o tamanho da matriz é especificado após o nome da variável, não após o nome do tipo, como em algumas outras linguagens. O exemplo a seguir declara uma matriz de 1000 duplos a serem alocados na pilha. O número de elementos deve ser fornecido como um literal inteiro ou, senão, como uma expressão constante. Isso ocorre porque o compilador precisa saber quanto espaço de pilha deve ser alocado; Ele não pode usar um valor calculado em tempo de execução. Cada elemento na matriz recebe um valor padrão de 0. Se você não atribuir um valor padrão, cada elemento inicialmente conterá quaisquer valores aleatórios que estejam nesse local de memória.

```
constexpr size_t size = 1000;

// Declare an array of doubles to be allocated on the stack
double numbers[size] {0};

// Assign a new value to the first element
numbers[0] = 1;

// Assign a value to each subsequent element
// (numbers[1] is the second element in the array.)
for (size_t i = 1; i < size; i++)
{
    numbers[i] = numbers[i-1] * 1.1;
}

// Access each element
for (size_t i = 0; i < size; i++)
{
    std::cout << numbers[i] << " ";
```

O primeiro elemento na matriz é o elemento inicial. O último elemento é o elemento $(n-1)$, em que n é o número de elementos que a matriz pode conter. O número de elementos na declaração deve ser de um tipo integral e deve ser maior que 0. É sua responsabilidade garantir que seu programa nunca passe um valor para o operador subscrito maior que `(size - 1)`.

Uma matriz de tamanho zero é válida somente quando a matriz é o último campo em um `struct` ou `union` e quando as extensões da Microsoft são habilitadas (`/za` ou `/permissive-` não é definido).

As matrizes baseadas em pilha são mais rápidas de alocar e acessar do que as matrizes baseadas em heap. No entanto, o espaço de pilha é limitado. O número de elementos de matriz não pode ser tão grande que usa muita memória de pilha. A quantidade de excesso de dependências do seu programa. Você pode usar ferramentas de criação de perfil para determinar se uma matriz é muito grande.

Declarações de heap

Você pode exigir uma matriz muito grande para alocar na pilha ou cujo tamanho não é conhecido no momento da compilação. É possível alocar essa matriz no heap usando uma `new[]` expressão. O operador retorna um ponteiro para o primeiro elemento. O operador subscrito funciona na variável de ponteiro da mesma maneira que em uma matriz baseada em pilha. Você também pode usar a [aritmética de ponteiro](#) para mover o ponteiro para qualquer elemento arbitrário na matriz. É sua responsabilidade garantir que:

- Você sempre mantém uma cópia do endereço do ponteiro original para poder excluir a memória quando não precisar mais da matriz.
- Você não incrementa ou Decrementa o endereço do ponteiro além dos limites da matriz.

O exemplo a seguir mostra como definir uma matriz no heap em tempo de execução. Ele mostra como acessar os elementos da matriz usando o operador subscrito e usando aritmética de ponteiro:

```

void do_something(size_t size)
{
    // Declare an array of doubles to be allocated on the heap
    double* numbers = new double[size]{ 0 };

    // Assign a new value to the first element
    numbers[0] = 1;

    // Assign a value to each subsequent element
    // (numbers[1] is the second element in the array.)
    for (size_t i = 1; i < size; i++)
    {
        numbers[i] = numbers[i - 1] * 1.1;
    }

    // Access each element with subscript operator
    for (size_t i = 0; i < size; i++)
    {
        std::cout << numbers[i] << " ";
    }

    // Access each element with pointer arithmetic
    // Use a copy of the pointer for iterating
    double* p = numbers;

    for (size_t i = 0; i < size; i++)
    {
        // Dereference the pointer, then increment it
        std::cout << *p++ << " ";
    }

    // Alternate method:
    // Reset p to numbers[0]:
    p = numbers;

    // Use address of pointer to compute bounds.
    // The compiler computes size as the number
    // of elements * (bytes per element).
    while (p < (numbers + size))
    {
        // Dereference the pointer, then increment it
        std::cout << *p++ << " ";
    }

    delete[] numbers; // don't forget to do this!
}

int main()
{
    do_something(108);
}

```

Inicializando matrizes

Você pode inicializar uma matriz em um loop, um elemento por vez ou em uma única instrução. O conteúdo das duas matrizes a seguir é idêntico:

```
int a[10];
for (int i = 0; i < 10; ++i)
{
    a[i] = i + 1;
}

int b[10]{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
```

Passando matrizes para funções

Quando uma matriz é passada para uma função, ela é passada como um ponteiro para o primeiro elemento, seja uma matriz baseada em pilha ou em heap. O ponteiro não contém informações adicionais de tamanho ou tipo. Esse comportamento é chamado de *ponteiro decaimento*. Ao passar uma matriz para uma função, você sempre deve especificar o número de elementos em um parâmetro separado. Esse comportamento também implica que os elementos da matriz não são copiados quando a matriz é passada para uma função. Para impedir que a função modifique os elementos, especifique o parâmetro como um ponteiro para `const` elementos.

O exemplo a seguir mostra uma função que aceita uma matriz e um comprimento. O ponteiro aponta para a matriz original, não para uma cópia. Como o parâmetro não é `const`, a função pode modificar os elementos da matriz.

```
void process(double p*, const size_t len)
{
    std::cout << "process:\n";
    for (size_t i = 0; i < len; ++i)
    {
        // do something with p[i]
    }
}
```

Declare a matriz como `const` para torná-la somente leitura dentro do bloco de função:

```
void process(const double p*, const size_t len);
```

A mesma função também pode ser declarada de maneiras, sem alterações no comportamento. A matriz ainda é passada como um ponteiro para o primeiro elemento:

```
// Unsized array
void process(const double p[], const size_t len);

// Fixed-size array. Length must still be specified explicitly.
void process(const double p[1000], const size_t len);
```

Matrizes multidimensionais

Matrizes construídas a partir de outras matrizes são matrizes multidimensionais. Essas matrizes multidimensionais são especificadas colocando várias expressões de constante entre colchetes em sequência. Por exemplo, considere esta declaração:

```
int i2[5][7];
```

Ele especifica uma matriz do tipo `int`, conceitualmente organizada em uma matriz bidimensional de cinco linhas e sete colunas, conforme mostrado na figura a seguir:

0, 0	0, 1	0, 2	0, 3	0, 4	0, 5	0, 6
1, 0	1, 1	1, 2	1, 3	1, 4	1, 5	1, 6
2, 0	2, 1	2, 2	2, 3	2, 4	2, 5	2, 6
3, 0	3, 1	3, 2	3, 3	3, 4	3, 5	3, 6
4, 0	4, 1	4, 2	4, 3	4, 4	4, 5	4, 6

Layout conceitual de uma matriz multidimensional

Você pode declarar matrizes multidimensionais que têm uma lista de inicializadores (conforme descrito em [inicializadores](#)). Nessas declarações, a expressão constante que especifica os limites para a primeira dimensão pode ser omitida. Por exemplo:

```
// arrays2.cpp
// compile with: /c
const int cMarkets = 4;
// Declare a float that represents the transportation costs.
double TransportCosts[][][cMarkets] = {
    { 32.19, 47.29, 31.99, 19.11 },
    { 11.29, 22.49, 33.47, 17.29 },
    { 41.97, 22.09, 9.76, 22.55 }
};
```

A declaração anterior define uma matriz com três linhas por quatro colunas. As linhas representam fábricas e colunas representam os mercados para os quais as fábricas enviam. Os valores são os custos de transporte das fábricas para os mercados. A primeira dimensão da matriz é deixada de fora, mas o compilador a preenche examinando o inicializador.

O uso do operador de indireção (*) em um tipo de matriz n-dimensional produz uma matriz dimensional n-1. Se n for 1, um escalar (ou elemento de matriz) será gerado.

As matrizes de C++ são armazenadas na ordem de linha principal. Ordem de linha principal significa que o último subscrito varia mais rápido.

Exemplo

Você também pode omitir a especificação de limites para a primeira dimensão de uma matriz multidimensional em declarações de função, como mostrado aqui:

```

// multidimensional_arrays.cpp
// compile with: /EHsc
// arguments: 3
#include <limits>    // Includes DBL_MAX
#include <iostream>

const int cMkts = 4, cFacts = 2;

// Declare a float that represents the transportation costs
double TransportCosts[][][cMkts] = {
    { 32.19, 47.29, 31.99, 19.11 },
    { 11.29, 22.49, 33.47, 17.29 },
    { 41.97, 22.09, 9.76, 22.55 }
};

// Calculate size of unspecified dimension
const int cFactories = sizeof TransportCosts /
    sizeof( double[cMkts] );

double FindMinToMkt( int Mkt, double myTransportCosts[][cMkts], int mycFacts);

using namespace std;

int main( int argc, char *argv[] ) {
    double MinCost;

    if (argv[1] == 0) {
        cout << "You must specify the number of markets." << endl;
        exit(0);
    }
    MinCost = FindMinToMkt( *argv[1] - '0', TransportCosts, cFacts);
    cout << "The minimum cost to Market " << argv[1] << " is: "
        << MinCost << "\n";
}

double FindMinToMkt(int Mkt, double myTransportCosts[][cMkts], int mycFacts) {
    double MinCost = DBL_MAX;

    for( size_t i = 0; i < cFacts; ++i )
        MinCost = (MinCost < TransportCosts[i][Mkt]) ?
            MinCost : TransportCosts[i][Mkt];

    return MinCost;
}

```

The minimum cost to Market 3 is: 17.29

A função `FindMinToMkt` é escrita de modo que a adição de novas fábricas não exija nenhuma alteração de código, apenas uma recompilação.

Inicializando matrizes

Matrizes de objetos que têm um construtor de classe são inicializadas pelo construtor. Quando há menos itens na lista de inicializadores do que elementos na matriz, o construtor padrão é usado para os elementos restantes. Se nenhum construtor padrão for definido para a classe, a lista de inicializadores deverá ser *concluída*, ou seja, deve haver um inicializador para cada elemento na matriz.

Considere a classe `Point` que define dois construtores:

```

// initializing_arrays1.cpp
class Point
{
public:
    Point() // Default constructor.
    {
    }
    Point( int, int ) // Construct from two ints
    {
    }
};

// An array of Point objects can be declared as follows:
Point aPoint[3] = {
    Point( 3, 3 ) // Use int, int constructor.
};

int main()
{
}

```

O primeiro elemento de `aPoint` é construído usando o construtor `Point(int, int)`; os dois elementos restantes são construídos com o construtor padrão.

Matrizes de membros estáticos (se `const` ou não) podem ser inicializadas em suas definições (fora da declaração de classe). Por exemplo:

```

// initializing_arrays2.cpp
class WindowColors
{
public:
    static const char *rgszWindowPartList[7];
};

const char *WindowColors::rgszWindowPartList[7] = {
    "Active Title Bar", "Inactive Title Bar", "Title Bar Text",
    "Menu Bar", "Menu Bar Text", "Window Background", "Frame"  };
int main()
{
}

```

Acessando elementos de matriz

Você pode acessar elementos individuais de uma matriz usando o operador de subscrito de matriz (`[]`). Se você usar o nome de uma matriz unidimensional sem um subscrito, ela será avaliada como um ponteiro para o primeiro elemento da matriz.

```

// using_arrays.cpp
int main() {
    char chArray[10];
    char *pch = chArray; // Evaluates to a pointer to the first element.
    char ch = chArray[0]; // Evaluates to the value of the first element.
    ch = chArray[3]; // Evaluates to the value of the fourth element.
}

```

Ao usar matrizes multidimensionais, você pode usar diversas combinações em expressões.

```

// using_arrays_2.cpp
// compile with: /EHsc /W1
#include <iostream>
using namespace std;
int main() {
    double multi[4][4][3]; // Declare the array.
    double (*p2multi)[3];
    double (*p1multi);

    cout << multi[3][2][2] << "\n"; // C4700 Use three subscripts.
    p2multi = multi[3]; // Make p2multi point to
                        // fourth "plane" of multi.
    p1multi = multi[3][2]; // Make p1multi point to
                        // fourth plane, third row
                        // of multi.
}

```

No código anterior, `multi` é uma matriz tridimensional do tipo `double`. O `p2multi` ponteiro aponta para uma matriz de tipo `double` de tamanho três. Nesse exemplo, a matriz é usada com um, dois e três subscritos. Embora seja mais comum especificar todos os subscritores, como na `cout` instrução, às vezes é útil selecionar um subconjunto específico de elementos de matriz, conforme mostrado nas instruções a seguir `cout`.

Sobrecarregando operador subscrito

Assim como outros operadores, o operador subscrito (`[]`) pode ser redefinido pelo usuário. O comportamento padrão do operador subscrito, se não sobrecarregado, é combinar o nome da matriz e o subscrito usando o seguinte método:

```
*((array_name) + (subscript))
```

Como em todas as adição que envolvem tipos de ponteiro, o dimensionamento é feito automaticamente para ajustar o tamanho do tipo. O valor resultante não é n bytes da origem de `array_name`; em vez disso, é o n -ésimo do elemento da matriz. Para obter mais informações sobre essa conversão, consulte [operadores aditivos](#).

De maneira semelhante, para matrizes multidimensionais, o endereço é derivado usando o seguinte método:

```
((array_name) + (subscript1 * max2 * max3 * ... * maxn) + (subscript2 * max3 * ... * maxn) + ... + subscriptn))
```

Matrizes em expressões

Quando um identificador de um tipo de matriz é exibido em uma expressão diferente de `sizeof`, endereço-de (`&`) ou inicialização de uma referência, ele é convertido em um ponteiro para o primeiro elemento da matriz. Por exemplo:

```

char szError1[] = "Error: Disk drive not ready.";
char *psz = szError1;

```

O ponteiro `psz` aponta para o primeiro elemento da matriz `szError1`. As matrizes, ao contrário de ponteiros, não são valores modificáveis. É por isso que a atribuição a seguir é ilegal:

```
szError1 = psz;
```

Confira também

[std::array](#)

Referências (C++)

02/09/2020 • 3 minutes to read • [Edit Online](#)

Uma referência, como um ponteiro, armazena o endereço de um objeto localizado em outro lugar na memória. Ao contrário de um ponteiro, uma referência após ser inicializada não pode ser feita para fazer referência a um objeto diferente ou definir como NULL. Há dois tipos de referências: referências lvalue que se referem a uma variável nomeada e referências a rvalue que se referem a um [objeto temporário](#). O operador `&` significa uma referência lvalue e o operador `&&` significa uma referência rvalue ou uma referência universal (rvalue ou lvalue), dependendo do contexto.

As referências podem ser declaradas usando a seguinte sintaxe:

*[especificadores de classe de armazenamento] [CV-qualificadores] tipo-especificadores [MS-modificador]
expressão de Declarador [= expression];*

Qualquer declarador válido que especifique uma referência pode ser usado. A menos que se trate de uma referência para um tipo de matriz ou função, a seguinte sintaxe simplificada se aplica:

[especificadores de classe de armazenamento] [CV-Qualifiers] tipo-especificadores [& ou &&] [CV-Qualifiers] expressão de identificador [= expression];

As referências são declaradas usando a seguinte sequência:

1. Os especificadores da declaração:

- Um especificador de classe de armazenamento opcional.
- `const` Qualificadores e/ou opcionais `volatile`.
- O especificador de tipo: o nome de um tipo.

2. O declarador:

- Um modificador opcional específico da Microsoft. Para obter mais informações, consulte [modificadores específicos da Microsoft](#).
- O `&` operador or `&&`.
- Opcional `const` e/ou `volatile` qualificadores.
- O identificador.

3. Um inicializador opcional.

Os formulários declaradores mais complexos para ponteiros para matrizes e funções também se aplicam a referências a matrizes e funções. Para obter mais informações, consulte [ponteiros](#).

Vários declaradores e inicializadores podem aparecer em uma lista separada por vírgulas após um único especificador de declaração. Por exemplo:

```
int &i;  
int &i, &j;
```

As referências, os ponteiros e os objetos podem ser declarados juntos:

```
int &ref, *ptr, k;
```

Uma referência contém o endereço de um objeto, mas se comporta sintaticamente como um objeto.

No programa a seguir, observe que o nome do objeto, `s`, e a referência ao objeto, `SRef`, podem ser usados de forma idêntica em programas:

Exemplo

```
// references.cpp
#include <stdio.h>
struct S {
    short i;
};

int main() {
    S s;      // Declare the object.
    S& SRef = s;  // Declare the reference.
    s.i = 3;

    printf_s("%d\n", s.i);
    printf_s("%d\n", SRef.i);

    SRef.i = 4;
    printf_s("%d\n", s.i);
    printf_s("%d\n", SRef.i);
}
```

```
3
3
4
4
```

Confira também

[Argumentos da função de tipo de referência](#)

[Função de tipo de referência retorna](#)

[Referências a ponteiros](#)

Declarador de referência lvalue:&

02/09/2020 • 2 minutes to read • [Edit Online](#)

Contém o endereço de um objeto mas se comporta sintaticamente como um objeto.

Sintaxe

```
type-id & cast-expression
```

Comentários

Você pode pensar em uma referência de lvalue como outro nome para um objeto. Uma declaração de referência de lvalue consiste de uma lista opcional de especificadores seguidos por um declarador de referência. Uma referência deve ser inicializada e não pode ser alterada.

Qualquer objeto cujo endereço possa ser convertido em um determinado tipo de ponteiro também pode ser convertido no tipo semelhante de referência. Por exemplo, qualquer objeto cujo endereço possa ser convertido para o tipo `char *` também pode ser convertido para o tipo `char &`.

Não confunda declarações de referência com o uso do [operador address-of](#). Quando o `& identificador` é precedido por um tipo, como `int` ou `char`, o *identificador* é declarado como uma referência ao tipo. Quando `&` o *identificador* não é precedido por um tipo, o uso é o do operador de endereço.

Exemplo

O exemplo a seguir demonstra o declarador de referência declarando um objeto `Person` e uma referência a esse objeto. Como `rFriend` é uma referência a `myFriend`, atualizar qualquer variável altera o mesmo objeto.

```
// reference_declarator.cpp
// compile with: /EHsc
// Demonstrates the reference declarator.
#include <iostream>
using namespace std;

struct Person
{
    char* Name;
    short Age;
};

int main()
{
    // Declare a Person object.
    Person myFriend;

    // Declare a reference to the Person object.
    Person& rFriend = myFriend;

    // Set the fields of the Person object.
    // Updating either variable changes the same object.
    myFriend.Name = "Bill";
    rFriend.Age = 40;

    // Print the fields of the Person object to the console.
    cout << rFriend.Name << " is " << myFriend.Age << endl;
}
```

```
Bill is 40
```

Confira também

[Referências](#)

[Argumentos da função de tipo de referência](#)

[Função de tipo de referência retorna](#)

[Referências a ponteiros](#)

Declarador de referência rvalue: &&

02/09/2020 • 22 minutes to read • [Edit Online](#)

Contém uma referência a uma expressão rvalue.

Sintaxe

```
type-id && cast-expression
```

Comentários

Referências de rvalue permitem que você diferencie um lvalue de um rvalue. As referências a lvalue e as referências a rvalue são sintaticamente e semanticamente semelhantes, mas seguem regras um pouco diferentes. Para obter mais informações sobre lvalues e rvalues, consulte [lvalues e rvalues](#). Para obter mais informações sobre referências de lvalue, consulte [Declarador de referência lvalue: &](#).

As seções a seguir descrevem como as referências a rvalue dão suporte à implementação de *semântica de movimentação e encaminhamento perfeito*.

Semântica de movimentação

As referências a rvalue dão suporte à implementação da *semântica de movimentação*, o que pode aumentar significativamente o desempenho de seus aplicativos. A semântica de movimentação permite que você escreva códigos que transfiram recursos (como a memória dinamicamente alocada) de um objeto para outro. A semântica de movimentação funciona porque permite que recursos sejam transferidos de objetos temporários que não podem ser referenciados em outro lugar no programa.

Para implementar a semântica de movimentação, você normalmente fornece um *Construtor de movimentação* e, opcionalmente, um operador de atribuição de movimento (**operador =**), para sua classe. As operações de cópia e atribuição cujas origens são rvalues aproveitam automaticamente as vantagens da semântica de movimentação. Diferente do construtor de cópia padrão, o compilador não fornece um construtor de movimentação padrão. Para obter mais informações sobre como escrever um construtor de movimentação e como usá-lo em seu aplicativo, consulte [mover construtores e mover operadores de atribuição \(C++\)](#).

Você também pode sobrepor as funções e operadores comuns para aproveitar a semântica de movimentação. O Visual Studio 2010 introduz a semântica de movimentação na biblioteca padrão C++. Por exemplo, a classe `string` implementa as operações que executam a semântica de movimentação. Considere o exemplo a seguir, que concatena várias cadeias de caracteres e imprime o resultado:

```
// string_concatenation.cpp
// compile with: /EHsc
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string s = string("h") + "e" + "ll" + "o";
    cout << s << endl;
}
```

Antes do Visual Studio 2010, cada chamada para **Operator +** aloca e retorna um novo objeto temporário `string` (um `rvalue`). O **operador +** não pode acrescentar uma cadeia de caracteres à outra porque não sabe se as cadeias de caracteres de origem são `lvalue` ou `rvalues`. Se as cadeias de caracteres de origem forem `lvalues`, poderão ser referenciadas em outro local do programa e, portanto, não devem ser modificadas. Usando referências `rvalue`, o **operador +** pode ser modificado para pegar `rvalues`, que não podem ser referenciados em outro lugar no programa. Portanto, o **operador +** pode agora acrescentar uma cadeia de caracteres a outra. Isso pode reduzir significativamente o número de alocações de memória dinâmica que a classe `string` deve executar. Para obter mais informações sobre a `string` classe, consulte [basic_string classe](#).

A semântica de movimentação também ajuda quando o compilador não pode usar a Otimização de Valor de Retorno (RVO) ou a Otimização de Valor de Retorno (NRVO). Nesses casos, o compilador chama o construtor de movimentação caso o tipo o defina.

Para compreender melhor a semântica de movimentação, considere o exemplo de inserção de um elemento em um objeto `vector`. Se a capacidade do objeto `vector` for excedida, o objeto `vector` deverá realocar memória para os seus elementos e então copiar cada elemento para outro local da memória para dar espaço ao elemento inserido. Quando uma operação de inserção copia um elemento, ele cria um novo elemento, chama o construtor de cópia para copiar os dados do elemento anterior para o novo elemento e destrói o elemento anterior. A semântica de movimentação permite que você move objetos diretamente sem ter que executar operações caras de alocação de memória e de cópia.

Para aproveitar a semântica de movimentação no exemplo `vector`, você pode escrever um construtor de movimentação para mover dados de um objeto para outro.

Para obter mais informações sobre a introdução da semântica de movimentação na biblioteca padrão do C++ no Visual Studio 2010, consulte [biblioteca padrão do c++](#).

Encaminhamento perfeito

O encaminhamento perfeito reduz a necessidade de funções sobrecarregadas e ajuda a evitar problemas de encaminhamento. O *problema de encaminhamento* pode ocorrer quando você escreve uma função genérica que usa referências como seus parâmetros e passa (ou *encaminha*) esses parâmetros para outra função. Por exemplo, se a função genérica receber um parâmetro de tipo `const T&`, então a função chamada não poderá modificar o valor desse parâmetro. Se a função genérica pegar um parâmetro do tipo `T&`, então a função não poderá ser chamada usando um `rvalue` (como um objeto temporário ou um literal de inteiro).

Normalmente, para resolver esse problema, você deve fornecer as versões sobrecarregadas da função genérica que recebe `T&` e `const T&` para cada um dos seus parâmetros. Como resultado, o número de funções sobrecarregadas aumenta exponencialmente com o número de parâmetros. As referências `rvalue` permitem que você grave uma versão de uma função que aceita argumentos arbitrários e os encaminha para outra função, como se a outra função tivesse sido chamada diretamente.

Considere o seguinte exemplo que declara quatro tipos, `w`, `x`, `y` e `z`. O construtor para cada tipo usa uma combinação diferente de `const` referências e não `const` `lvalue` como seus parâmetros.

```

struct W
{
    W(int&, int&) {}
};

struct X
{
    X(const int&, int&) {}
};

struct Y
{
    Y(int&, const int&) {}
};

struct Z
{
    Z(const int&, const int&) {}
};

```

Suponha que você quer gravar uma função genérica que gera objetos. O exemplo a seguir mostra uma maneira de gravar esta função:

```

template <typename T, typename A1, typename A2>
T* factory(A1& a1, A2& a2)
{
    return new T(a1, a2);
}

```

O exemplo a seguir mostra uma chamada válida para a função `factory`:

```

int a = 4, b = 5;
W* pw = factory<W>(a, b);

```

No entanto, o exemplo a seguir não contém uma chamada válida para a função `factory` porque `factory` pega referências lvalue que são modificáveis como seus parâmetros, mas é chamada usando rvalues:

```

Z* pz = factory<Z>(2, 2);

```

Normalmente, para resolver esse problema, você deve criar uma versão sobrecarregada da função `factory` para cada combinação de parâmetros `A&` e `const A&`. As referências de rvalue permitem que você grave uma versão da função `factory`, conforme mostrado no exemplo o seguir:

```

template <typename T, typename A1, typename A2>
T* factory(A1&& a1, A2&& a2)
{
    return new T(std::forward<A1>(a1), std::forward<A2>(a2));
}

```

Este exemplo usa referências de rvalue como os parâmetros para a função `factory`. A finalidade da função `std::forward` é encaminhar os parâmetros da função de fábrica para o construtor da classe de modelo.

O exemplo a seguir mostra a função `main` que usa a função `factory` revisada para criar instâncias das classes `W`, `X`, `Y` e `Z`. A função `factory` revisada encaminha seus parâmetros (lvalues ou rvalues) para o construtor de classe apropriado.

```

int main()
{
    int a = 4, b = 5;
    W* pw = factory<W>(a, b);
    X* px = factory<X>(2, b);
    Y* py = factory<Y>(a, 2);
    Z* pz = factory<Z>(2, 2);

    delete pw;
    delete px;
    delete py;
    delete pz;
}

```

Propriedades adicionais das referências de rvalue

Você pode sobrestrar uma função para obter uma referência de lvalue e uma referência de rvalue.

Ao sobrestrar uma função para obter uma `const` referência lvalue ou uma referência rvalue, você pode escrever código que diferencie entre objetos não modificáveis (lvalues) e valores temporários modificáveis (rvalues). Você pode passar um objeto para uma função que usa uma referência rvalue, a menos que o objeto seja marcado como `const`. O exemplo a seguir mostra a função `f`, que é sobrestrada para pegar uma referência de lvalue e uma referência de rvalue. A função `main` chama `f` com lvalues e um rvalue.

```

// reference-overload.cpp
// Compile with: /EHsc
#include <iostream>
using namespace std;

// A class that contains a memory resource.
class MemoryBlock
{
    // TODO: Add resources for the class here.
};

void f(const MemoryBlock&)
{
    cout << "In f(const MemoryBlock&). This version cannot modify the parameter." << endl;
}

void f(MemoryBlock&&)
{
    cout << "In f(MemoryBlock&&). This version can modify the parameter." << endl;
}

int main()
{
    MemoryBlock block;
    f(block);
    f(MemoryBlock());
}

```

Esse exemplo gera a saída a seguir:

```

In f(const MemoryBlock&). This version cannot modify the parameter.
In f(MemoryBlock&&). This version can modify the parameter.

```

Nesse exemplo, a primeira chamada para `f` passa uma variável local (um lvalue) como seu argumento. A

segunda chamada para `f` passa um objeto temporário como seu argumento. Como o objeto temporário não pode ser referenciado em outro lugar no programa, a chamada é associada à versão sobre carregada de `f` que utiliza uma referência de rvalue, que está livre para alterar o objeto.

O compilador trata uma referência de rvalue nomeada como um lvalue e uma referência de rvalue não nomeada como um rvalue.

Quando você escrever uma função que recebe uma referência de rvalue como seu parâmetro, esse parâmetro é tratado como um lvalue no corpo da função. O compilador trata uma referência de rvalue nomeada como um lvalue porque um objeto nomeado pode ser referenciado por várias partes de um programa. Seria perigoso permitir que várias partes de um programa modifiquem ou removam recursos desse objeto. Por exemplo, se várias partes de um programa tentarem transferir recursos do mesmo objeto, somente a primeira parte transferirá o recurso com êxito.

O exemplo a seguir mostra a função `g`, que é sobre carregada para pegar uma referência de lvalue e uma referência de rvalue. A função `f` pega uma referência de rvalue como seu parâmetro (uma referência de rvalue nomeada) e retorna uma referência de rvalue (uma referência de rvalue sem nome). Na chamada a `g` de `f`, a resolução de sobre carregamento seleciona a versão de `g` que utiliza uma referência de lvalue porque o corpo de `f` trata seu parâmetro como um lvalue. Na chamada a `g` de `main`, a resolução de sobre carregamento seleciona a versão de `g` que utiliza uma referência de rvalue porque `f` retorna uma referência de rvalue.

```
// named-reference.cpp
// Compile with: /EHsc
#include <iostream>
using namespace std;

// A class that contains a memory resource.
class MemoryBlock
{
    // TODO: Add resources for the class here.
};

void g(const MemoryBlock&)
{
    cout << "In g(const MemoryBlock&)." << endl;
}

void g(MemoryBlock&&)
{
    cout << "In g(MemoryBlock&&)." << endl;
}

MemoryBlock&& f(MemoryBlock&& block)
{
    g(block);
    return move(block);
}

int main()
{
    g(f(MemoryBlock()));
}
```

Esse exemplo gera a saída a seguir:

```
In g(const MemoryBlock&).
In g(MemoryBlock&&).
```

Neste exemplo, a função `main` passa um rvalue para `f`. O corpo de `f` trata seu parâmetro nomeado como um

lvalue. A chamada de `f` para `g` associa o parâmetro a uma referência de lvalue (a primeira versão sobrecregada de `g`).

- Você pode converter um lvalue em uma referência rvalue.

A função padrão de biblioteca do C++ `std::move` permite converter um objeto em uma referência rvalue para esse objeto. Como alternativa, você pode usar a `static_cast` palavra-chave para converter um lvalue em uma referência rvalue, conforme mostrado no exemplo a seguir:

```
// cast-reference.cpp
// Compile with: /EHsc
#include <iostream>
using namespace std;

// A class that contains a memory resource.
class MemoryBlock
{
    // TODO: Add resources for the class here.
};

void g(const MemoryBlock&)
{
    cout << "In g(const MemoryBlock&)." << endl;
}

void g(MemoryBlock&&)
{
    cout << "In g(MemoryBlock&&)." << endl;
}

int main()
{
    MemoryBlock block;
    g(block);
    g(static_cast<MemoryBlock&&>(block));
}
```

Esse exemplo gera a saída a seguir:

```
In g(const MemoryBlock&).
In g(MemoryBlock&&).
```

Os modelos de função deduzem seus tipos de argumento de modelo e então usam regras de recolhimento de referência.

É comum escrever um modelo de função que passe (ou *encaminhe*) seus parâmetros para outra função. É importante compreender como funciona a dedução de tipo de modelo para os modelos de função com referências de rvalue.

Se o argumento de função for um rvalue, o compilador deduzirá o argumento como uma referência de rvalue. Por exemplo, se você passar uma referência de rvalue para um objeto do tipo `x` a uma função de modelo que obtenha o tipo `T&&` como seu parâmetro, a dedução do argumento do modelo deduzirá `T` como `x`. Portanto, o parâmetro tem o tipo `x&&`. Se o argumento da função for um lvalue ou `const` lvalue, o compilador deduzirá seu tipo para ser uma referência lvalue ou uma `const` referência lvalue desse tipo.

O exemplo a seguir declara um modelo de estrutura e o especializa para vários tipos de referência. A função `print_type_and_value` usa uma referência de rvalue como seu parâmetro e a encaminha para a versão especializada apropriada do método `S::print`. A função `main` demonstra as várias maneiras de chamar o método `S::print`.

```

// template-type-deduction.cpp
// Compile with: /EHsc
#include <iostream>
#include <string>
using namespace std;

template<typename T> struct S {

    // The following structures specialize S by
    // lvalue reference (T&), const lvalue reference (const T&),
    // rvalue reference (T&&), and const rvalue reference (const T&&).
    // Each structure provides a print method that prints the type of
    // the structure and its parameter.

    template<typename T> struct S<T&> {
        static void print(T& t)
        {
            cout << "print<T&>: " << t << endl;
        }
    };

    template<typename T> struct S<const T&> {
        static void print(const T& t)
        {
            cout << "print<const T&>: " << t << endl;
        }
    };

    template<typename T> struct S<T&&> {
        static void print(T&& t)
        {
            cout << "print<T&&>: " << t << endl;
        }
    };

    template<typename T> struct S<const T&&> {
        static void print(const T&& t)
        {
            cout << "print<const T&&>: " << t << endl;
        }
    };

    // This function forwards its parameter to a specialized
    // version of the S type.
    template <typename T> void print_type_and_value(T&& t)
    {
        S<T&&>::print(std::forward<T>(t));
    }

    // This function returns the constant string "fourth".
    const string fourth() { return string("fourth"); }

    int main()
    {
        // The following call resolves to:
        // print_type_and_value<string&>(string& && t)
        // Which collapses to:
        // print_type_and_value<string&>(string& t)
        string s1("first");
        print_type_and_value(s1);

        // The following call resolves to:
        // print_type_and_value<const string&>(const string& && t)
        // Which collapses to:
        // print_type_and_value<const string&>(const string& t)
        const string s2("second");
        print_type_and_value(s2);
    }
}

```

```

// The following call resolves to:
// print_type_and_value<string&&>(string&& t)
print_type_and_value(string("third"));

// The following call resolves to:
// print_type_and_value<const string&&>(const string&& t)
print_type_and_value(fourth());
}

```

Esse exemplo gera a saída a seguir:

```

print<T&>: first
print<const T&>: second
print<T&&>: third
print<const T&&>: fourth

```

Para resolver cada chamada para a função `print_type_and_value`, o compilador executa primeiro a dedução do argumento do modelo. O compilador, então, aplica regras de recolhimento de referência quando substitui os argumentos de modelo deduzidos para os tipos de parâmetro. Por exemplo, passar a variável local `s1` para a função `print_type_and_value` faz com que o compilador produza a seguinte assinatura de função:

```
print_type_and_value<string&>(string& && t)
```

O compilador usa regras de recolhimento de referência para reduzir a assinatura para:

```
print_type_and_value<string&>(string& t)
```

Esta versão da função `print_type_and_value` encaminha seu parâmetro para a versão especializada correta do método `S::print`.

A tabela a seguir resume as regras de recolhimento de referência para dedução do tipo de argumento de modelo:

TIPO EXPANDIDO	TIPO RECOLHIDO
<code>T& &</code>	<code>T&</code>
<code>T& &&</code>	<code>T&</code>
<code>T&& &</code>	<code>T&</code>
<code>T&& &&</code>	<code>T&&</code>

A dedução do argumento do modelo é um elemento importante para a implementação do encaminhamento perfeito. A seção Encaminhamento perfeito apresentada anteriormente neste tópico descreve o encaminhamento perfeito com mais detalhes.

Resumo

Referências de rvalue diferenciam lvalues de rvalues. Elas podem ajudar a melhorar o desempenho de seus aplicativos eliminando a necessidade de alocações de memória e operações de cópia desnecessárias. Elas também permitem que você grave uma versão de uma função que aceita argumentos arbitrários e os encaminha para outra função, como se a outra função tivesse sido chamada diretamente.

Confira também

[Expressões com operadores unários](#)

[Declarador de referência Lvalue: &](#)

[Lvalues e rvalues](#)

[Move Constructors and Move Assignment Operators \(C++\)](#) (Operadores de construtores de movimento e de atribuição de movimento (C++))

[Biblioteca padrão do C++](#)

Argumentos de funções de tipo de referência

02/09/2020 • 2 minutes to read • [Edit Online](#)

É geralmente mais eficiente passar referências, em vez de objetos grandes, para funções. Isso permite que o compilador passe o endereço do objeto enquanto mantém a sintaxe que seria usada para acessar o objeto. Considere o seguinte exemplo que usa a estrutura `Date`:

```
// reference_type_function_arguments.cpp
#include <iostream>

struct Date
{
    short Month;
    short Day;
    short Year;
};

// Create a date of the form DDDYYYY (day of year, year)
// from a Date.
long DateOfYear( Date& date )
{
    static int cDaysInMonth[] = {
        31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
    };
    long dateOfYear = 0;

    // Add in days for months already elapsed.
    for ( int i = 0; i < date.Month - 1; ++i )
        dateOfYear += cDaysInMonth[i];

    // Add in days for this month.
    dateOfYear += date.Day;

    // Check for leap year.
    if ( date.Month > 2 &&
        ( ( date.Year % 100 != 0 || date.Year % 400 == 0 ) &&
        date.Year % 4 == 0 ) )
        dateOfYear++;

    // Add in year.
    dateOfYear *= 10000;
    dateOfYear += date.Year;

    return dateOfYear;
}

int main()
{
    Date date{ 8, 27, 2018 };
    long dateOfYear = DateOfYear(date);
    std::cout << dateOfYear << std::endl;
}
```

O código anterior mostra que os membros de uma estrutura passada por referência são acessados usando o operador de seleção de membro `(.)` em vez do operador de seleção de membro de ponteiro `(->)`.

Embora os argumentos passados como tipos de referência observem a sintaxe dos tipos que não são ponteiros, eles retêm uma característica importante dos tipos de ponteiro: eles podem ser modificados, a menos que sejam declarados como `const`. Como a intenção do código anterior não é alterar o objeto `date`, um protótipo de

função mais apropriado é:

```
long DateOfYear( const Date& date );
```

Esse protótipo garante que a função `DateOfYear` não alterará seu argumento.

Qualquer função com protótipo como pegar um tipo de referência pode aceitar um objeto do mesmo tipo em seu lugar porque há uma conversão padrão de `TypeName` para `TypeName &`.

Confira também

[Referências](#)

Retornos de funções de tipo de referência

02/09/2020 • 4 minutes to read • [Edit Online](#)

As funções podem ser declaradas para retornar um tipo de referência. Há duas razões para fazer tal declaração:

- As informações retornadas são um objeto tão grande que retornar uma referência é mais eficiente do que retornar uma cópia.
- O tipo da função deve ser um l-value.
- O objeto referenciado não ficará fora do escopo quando a função retornar.

Assim como pode ser mais eficiente para passar objetos grandes *para* funções por referência, também pode ser mais eficiente retornar objetos grandes *de* funções por referência. O protocolo de retorno de referência elimina a necessidade de copiar o objeto em um local temporário antes de retornar.

Os tipos de retorno de referência também podem ser úteis quando a função deve ser avaliada como um l-value. A maioria dos operadores sobrecarregados entra nessa categoria, particularmente o operador de atribuição.

Operadores sobrecarregados são abordados em [operadores sobrecarregados](#).

Exemplo

Considere o exemplo de `Point` :

```

// refType_function_returns.cpp
// compile with: /EHsc

#include <iostream>
using namespace std;

class Point
{
public:
// Define "accessor" functions as
// reference types.
unsigned& x();
unsigned& y();

private:
// Note that these are declared at class scope:
unsigned obj_x;
unsigned obj_y;
};

unsigned& Point :: x()
{
return obj_x;
}

unsigned& Point :: y()
{
return obj_y;
}

int main()
{
Point ThePoint;
// Use x() and y() as l-values.
ThePoint.x() = 7;
ThePoint.y() = 9;

// Use x() and y() as r-values.
cout << "x = " << ThePoint.x() << "\n"
<< "y = " << ThePoint.y() << "\n";
}

```

Saída

```

x = 7
y = 9

```

Observe que as funções `x` e `y` são declaradas como tipos de referência de retorno. Essas funções podem ser usadas em ambos os lados de uma instrução de atribuição.

Observe também que, em Main, o objeto Point permanece no escopo e, portanto, seus membros de referência ainda estão ativos e podem ser acessados com segurança.

As declarações de tipos de referência devem conter inicializadores, exceto nos seguintes casos:

- `extern` Declaração explícita
- Declaração de um membro de classe
- Declaração em uma classe
- Declaração de um argumento para uma função ou o tipo de retorno para uma função

Cuidado ao retornar o endereço do local

Se você declarar um objeto no escopo local, esse objeto será destruído quando a função retornar. Se a função retornar uma referência a esse objeto, essa referência provavelmente causará uma violação de acesso em tempo de execução se o chamador tentar usar a referência nula.

```
// C4172 means Don't do this!!!
Foo& GetFoo()
{
    Foo f;
    ...
    return f;
} // f is destroyed here
```

O compilador emite um aviso nesse caso: `warning C4172: returning address of local variable or temporary`. Em programas simples, é possível que, ocasionalmente, nenhuma violação de acesso ocorra se a referência for acessada pelo chamador antes de o local da memória ser substituído. Isso ocorre devido à simples sorte. Preste atenção ao aviso.

Confira também

[Referências](#)

Referências a ponteiros

02/12/2019 • 3 minutes to read • [Edit Online](#)

As referências para ponteiros podem ser declaradas de forma semelhante às referências para objetos. Uma referência a um ponteiro é um valor modificável que é usado como um ponteiro normal.

Exemplo

Este exemplo de código mostra a diferença entre usar um ponteiro para um ponteiro e uma referência a um ponteiro.

Funções `Add1` e `Add2` são funcionalmente equivalentes, embora eles não são chamados da mesma maneira. A diferença é que `Add1` usa indireção double, mas `Add2` usa a conveniência de uma referência a um ponteiro.

```
// references_to_pointers.cpp
// compile with: /EHsc

#include <iostream>
#include <string>

// C++ Standard Library namespace
using namespace std;

enum {
    sizeOfBuffer = 132
};

// Define a binary tree structure.
struct BTree {
    char *szText;
    BTree *Left;
    BTree *Right;
};

// Define a pointer to the root of the tree.
BTree *btRoot = 0;

int Add1( BTree **Root, char *szToAdd );
int Add2( BTree*& Root, char *szToAdd );
void PrintTree( BTree* btRoot );

int main( int argc, char *argv[] ) {
    // Usage message
    if( argc < 2 ) {
        cerr << "Usage: " << argv[0] << " [1 | 2]" << "\n";
        cerr << "\nwhere:\n";
        cerr << "1 uses double indirection\n";
        cerr << "2 uses a reference to a pointer.\n";
        cerr << "\nInput is from stdin. Use ^Z to terminate input.\n";
        return 1;
    }

    char *szBuf = new char[sizeOfBuffer];
    if (szBuf == NULL) {
        cerr << "Out of memory!\n";
        return -1;
    }

    // Read a text file from the standard input device and
    // build a binary tree.
```

```

while( !cin.eof() )
{
    cin.get( szBuf, sizeOfBuffer, '\n' );
    cin.get();

    if ( strlen( szBuf ) ) {
        switch ( *argv[1] ) {
            // Method 1: Use double indirection.
            case '1':
                Add1( &btRoot, szBuf );
                break;
            // Method 2: Use reference to a pointer.
            case '2':
                Add2( btRoot, szBuf );
                break;
            default:
                cerr << "Illegal value "
                << *argv[1]
                << "' supplied for add method.\n"
                << "Choose 1 or 2.\n";
                return -1;
        }
    }
}

// Display the sorted list.
PrintTree( btRoot );
}

// PrintTree: Display the binary tree in order.
void PrintTree( BTTree* MybtRoot ) {
    // Traverse the left branch of the tree recursively.
    if ( MybtRoot->Left )
        PrintTree( MybtRoot->Left );

    // Print the current node.
    cout << MybtRoot->szText << "\n";

    // Traverse the right branch of the tree recursively.
    if ( MybtRoot->Right )
        PrintTree( MybtRoot->Right );
}

// Add1: Add a node to the binary tree.
//      Uses double indirection.
int Add1( BTTree **Root, char *szToAdd ) {
    if ( (*Root) == 0 ) {
        (*Root) = new BTTree;
        (*Root)->Left = 0;
        (*Root)->Right = 0;
        (*Root)->szText = new char[strlen( szToAdd ) + 1];
        strcpy_s((*Root)->szText, (strlen( szToAdd ) + 1), szToAdd );
        return 1;
    }
    else {
        if ( strcmp( (*Root)->szText, szToAdd ) > 0 )
            return Add1( &((*Root)->Left), szToAdd );
        else
            return Add1( &((*Root)->Right), szToAdd );
    }
}

// Add2: Add a node to the binary tree.
//      Uses reference to pointer
int Add2( BTTree*& Root, char *szToAdd ) {
    if ( Root == 0 ) {
        Root = new BTTree;
        Root->Left = 0;
        Root->Right = 0;
        Root->szText = new char[strlen( szToAdd ) + 1];
    }
}

```

```
    strcpy_s( Root->szText, (strlen( szToAdd ) + 1), szToAdd );
    return 1;
}
else {
    if ( strcmp( Root->szText, szToAdd ) > 0 )
        return Add2( Root->Left, szToAdd );
    else
        return Add2( Root->Right, szToAdd );
}
}
```

```
Usage: references_to_pointers.exe [1 | 2]
```

where:

1 uses double indirection

2 uses a reference to a pointer.

Input is from stdin. Use ^Z to terminate input.

Consulte também

[Referências](#)

Ponteiros (C++)

15/04/2020 • 2 minutes to read • [Edit Online](#)

Um ponteiro é uma variável que armazena o endereço de memória de um objeto. Os ponteiros são usados extensivamente em C e C++ para três propósitos principais:

- para alocar novos objetos na pilha,
- para passar funções para outras funções
- para iterar sobre elementos em matrizes ou outras estruturas de dados.

Na programação estilo C, *ponteiros brutos* são usados para todos esses cenários. No entanto, ponteiros brutos são a fonte de muitos erros sérios de programação. Portanto, seu uso é fortemente desencorajado, exceto quando eles fornecem um benefício significativo de desempenho e não há ambiguidade sobre qual ponteiro é o *ponteiro que* é responsável pela exclusão do objeto. O C++ moderno fornece *ponteiros inteligentes* para alocar *objetos*, *iteradores* para atravessar estruturas de dados e *expressões lambda* para funções de passagem. Ao usar essas instalações de idioma e biblioteca em vez de ponteiros brutos, você tornará seu programa mais seguro, mais fácil de depurar e mais simples de entender e manter. Consulte [ponteiros inteligentes, erros de vida e expressões Lambda](#) para obter mais informações.

Nesta seção

- [Ponteiros brutos](#)
- [Ponteiros const e voláteis](#)
- [novos e excluir operadores](#)
- [Ponteiros inteligentes](#)
- [Como: Criar e usar unique_ptr instâncias](#)
- [Como: Criar e usar shared_ptr instâncias](#)
- [Como: Criar e usar weak_ptr instâncias](#)
- [Como: Criar e usar as instâncias CComPtr e CComQIPtr](#)

Confira também

[Iterators](#)

[Expressões lambda](#)

Ponteiros brutos (C++)

02/09/2020 • 14 minutes to read • [Edit Online](#)

Um *ponteiro* é um tipo de variável. Ele armazena o endereço de um objeto na memória e é usado para acessar esse objeto. Um *ponteiro bruto* é um ponteiro cujo tempo de vida não é controlado por um objeto encapsulamento, como um *ponteiro inteligente*. Um ponteiro bruto pode ser atribuído ao endereço de outra variável que não seja de ponteiro ou pode ser atribuído a um valor de . Um ponteiro que não tenha sido atribuído a um valor contém dados aleatórios.

Um ponteiro também pode ser *desreferenciado* para recuperar o valor do objeto que ele aponta. O *operador de acesso de membro* fornece acesso aos membros de um objeto.

```
int* p = ::::no-loc(nullptr):::; // declare pointer and initialize it
                                // so that it doesn't store a random address
int i = 5;
p = &i; // assign pointer to address of object
int j = *p; // dereference p to retrieve the value at its address
```

Um ponteiro pode apontar para um objeto tipado ou para `::::no-loc(void):::` . Quando um programa aloca um objeto no *heap* na memória, ele recebe o endereço desse objeto na forma de um ponteiro. Tais ponteiros são chamados de *ponteiros de propriedade*. Um ponteiro de propriedade (ou uma cópia dele) deve ser usado para liberar explicitamente o objeto alocado de heap quando ele não for mais necessário. A falha na liberação da memória resulta em um *vazamento de memória* e renderiza esse local de memória indisponível para qualquer outro programa no computador. A memória alocada usando o `::::no-loc(new):::` deve ser liberada usando `::::no-loc(delete):::` (ou `** ::::no-loc(delete)::: []**`). Para obter mais informações, consulte .

```
MyClass* mc = ::::no-loc(new)::: MyClass(); // allocate object on the heap
mc->print(); // access class member
:::no-loc(delete)::: mc; // ::::no-loc(delete)::: object (please don't forget!)
```

Um ponteiro (se não for declarado como `::::no-loc(const):::`) pode ser incrementado ou diminuído para apontar para outro local na memória. Essa operação é chamada de *aritmética de ponteiro*. Ele é usado na programação C-Style para iterar sobre elementos em matrizes ou outras estruturas de dados. `::::no-loc(const):::` Não é possível estabelecer um ponteiro para apontar para um local de memória diferente e, nesse sentido, é semelhante a uma *referência*. Para obter mais informações, consulte .

```
// declare a C-style string. Compiler adds terminating '\0'.
:::no-loc(const)::: ::::no-loc(char):::* str = "Hello world";

:::no-loc(const)::: int c = 1;
:::no-loc(const)::: int* p:::no-loc(const)::: = &c; // declare a non-:::no-loc(const)::: pointer to ::::no-loc(const)::: int
:::no-loc(const)::: int c2 = 2;
p:::no-loc(const)::: = &c2; // OK p:::no-loc(const)::: itself isn't ::::no-loc(const):::
:::no-loc(const)::: int* ::::no-loc(const)::: p:::no-loc(const):::2 = &c;
// p:::no-loc(const):::2 = &c2; // Error! p:::no-loc(const):::2 is ::::no-loc(const):::.
```

Em sistemas operacionais de 64 bits, um ponteiro tem um tamanho de 64 bits. O tamanho de um ponteiro do sistema determina o quanto de memória endereçável ele pode ter. Todas as cópias de um ponteiro apontam para o mesmo local de memória. Ponteiros (juntamente com referências) são usados extensivamente em C++ para passar objetos maiores de e para funções. Isso porque geralmente é mais eficiente copiar o endereço de um objeto

do que copiar o objeto inteiro. Ao definir uma função, especifique parâmetros de ponteiro como

`::::no-loc(const)::::`, a menos que você pretenda a função para modificar o objeto. Em geral,

`::::no-loc(const)::::` as referências são a maneira preferida de passar objetos para funções, a menos que o valor do objeto possivelmente possa ser `::::no-loc(nullptr)::::`.

[Ponteiros para funções](#) permitem que as funções sejam passadas para outras funções e são usadas para "retornos de chamada" na programação em estilo C. O C++ moderno usa [expressões lambda](#) para essa finalidade.

Inicialização e acesso de membro

O exemplo a seguir mostra como declarar, inicializar e usar um ponteiro bruto. Ele é inicializado usando

`::::no-loc(new)::::` para apontar um objeto alocado no heap, que você deve explicitamente `::::no-loc(delete)::::`.

O exemplo também mostra alguns dos perigos associados a ponteiros brutos. (Lembre-se, este exemplo é a programação em estilo C e não C++ moderno!)

```
#include <iostream>
#include <string>

class MyClass
{
public:
    int num;
    std::string name;
    ::::no-loc(void):::: print() { std::cout << name << ":" << num << std::endl; }

};

// Accepts a MyClass pointer
::::no-loc(void):::: func_A(MyClass* mc)
{
    // Modify the object that mc points to.
    // All copies of the pointer will point to
    // the same modified object.
    mc->num = 3;
}

// Accepts a MyClass object
::::no-loc(void):::: func_B(MyClass mc)
{
    // mc here is a regular object, not a pointer.
    // Use the "." operator to access members.
    // This statement modifies only the local copy of mc.
    mc.num = 21;
    std::cout << "Local copy of mc:";
    mc.print(); // "Erika, 21"
}

int main()
{
    // Use the * operator to declare a pointer type
    // Use ::::no-loc(new):::: to allocate and initialize memory
    MyClass* pmc = ::::no-loc(new):::: MyClass{ 108, "Nick" };

    // Prints the memory address. Usually not what you want.
    std::cout << pmc << std::endl;

    // Copy the pointed-to object by dereferencing the pointer
    // to access the contents of the memory location.
    // mc is a separate object, allocated here on the stack
    MyClass mc = *pmc;

    // Declare a pointer that points to mc using the addressof operator
    MyClass* pcopy = &mc;
```

```

// use the -> operator to access the object's public members
pmc->print(); // "Nick, 108"

// Copy the pointer. Now pmc and pmc2 point to same object!
MyClass* pmc2 = pmc;

// Use copied pointer to modify the original object
pmc2->name = "Erika";
pmc->print(); // "Erika, 108"
pmc2->print(); // "Erika, 108"

// Pass the pointer to a function.
func_A(pmc);
pmc->print(); // "Erika, 3"
pmc2->print(); // "Erika, 3"

// Dereference the pointer and pass a copy
// of the pointed-to object to a function
func_B(*pmc);
pmc->print(); // "Erika, 3" (original not modified by function)

:::no-loc(delete):::(pmc); // don't forget to give memory back to operating system!
// ::::no-loc(delete):::(pmc2); //crash! memory location was already ::no-loc(delete):::
}

```

Aritmética de ponteiro e matrizes

Ponteiros e matrizes estão fortemente relacionados. Quando uma matriz é passada por valor para uma função, ela é passada como um ponteiro para o primeiro elemento. O exemplo a seguir demonstra as seguintes propriedades importantes de ponteiros e matrizes:

- O `sizeof` operador retorna o tamanho total em bytes de uma matriz
- para determinar o número de elementos, divida o total de bytes pelo tamanho de um elemento
- Quando uma matriz é passada para uma função, ela *decays* a um tipo de ponteiro
- O `sizeof` operador quando aplicado a um ponteiro retorna o tamanho do ponteiro, 4 bytes em x86 ou 8 bytes em x64

```

#include <iostream>

:::no-loc(void)::: func(int arr[], int length)
{
    // returns pointer size. not useful here.
    size_t test = sizeof(arr);

    for(int i = 0; i < length; ++i)
    {
        std::cout << arr[i] << " ";
    }
}

int main()
{
    int i[5]{ 1,2,3,4,5 };
    // sizeof(i) = total bytes
    int j = sizeof(i) / sizeof(i[0]);
    func(i,j);
}

```

Determinadas operações aritméticas podem ser usadas em não `::no-loc(const)::` ponteiros para torná-las apontadas para outro local de memória. Os ponteiros são incrementados e decrementados usando os `++` `+=` operadores, `-` `=` `--`. Essa técnica pode ser usada em matrizes e é especialmente útil em buffers de dados não

tipados. Um `:::no-loc(void):::*` é incrementado pelo tamanho de um `:::no-loc(char):::` (1 byte). Um ponteiro tipado é incrementado por tamanho do tipo para o qual ele aponta.

O exemplo a seguir demonstra como a aritmética de ponteiro pode ser usada para acessar pixels individuais em um bitmap no Windows. Observe o uso de `:::no-loc(new):::` e `:::no-loc(delete):::` o operador de desreferência.

```

#include <Windows.h>
#include <fstream>

using namespace std;

int main()
{
    BITMAPINFOHEADER header;
    header.biHeight = 100; // Multiple of 4 for simplicity.
    header.biWidth = 100;
    header.biBitCount = 24;
    header.biPlanes = 1;
    header.biCompression = BI_RGB;
    header.biSize = sizeof(BITMAPINFOHEADER);

    ::::no-loc(const):::expr int bufferSize = 30000;
    unsigned ::::no-loc(char):::* buffer = ::::no-loc(new)::: unsigned ::::no-loc(char):::[bufferSize];

    BITMAPFILEHEADER bf;
    bf.bfType = 0x4D42;
    bf.bfSize = header.biSize + 14 + bufferSize;
    bf.bfReserved1 = 0;
    bf.bfReserved2 = 0;
    bf.bfOffBits = sizeof(BITMAPFILEHEADER) + sizeof(BITMAPINFOHEADER); //54

    // Create a gray square with a 2-pixel wide outline.
    unsigned ::::no-loc(char):::* begin = &buffer[0];
    unsigned ::::no-loc(char):::* end = &buffer[0] + bufferSize;
    unsigned ::::no-loc(char):::* p = begin;
    ::::no-loc(const):::expr int pixelWidth = 3;
    ::::no-loc(const):::expr int borderWidth = 2;

    while (p < end)
    {
        // Is top or bottom edge?
        if ((p < begin + header.biWidth * pixelWidth * borderWidth)
            || (p > end - header.biWidth * pixelWidth * borderWidth))
            // Is left or right edge?
            || (p - begin) % (header.biWidth * pixelWidth) < (borderWidth * pixelWidth)
            || (p - begin) % (header.biWidth * pixelWidth) > ((header.biWidth - borderWidth) * pixelWidth))
        {
            *p = 0x0; // Black
        }
        else
        {
            *p = 0xC3; // Gray
        }
        p++; // Increment one byte sizeof(unsigned ::::no-loc(char):::).
    }

    ofstream wf(R"(box.bmp)", ios::out | ios::binary);

    wf.write(reinterpret_cast<:::no-loc(char):::*>(&bf), sizeof(bf));
    wf.write(reinterpret_cast<:::no-loc(char):::*>(&header), sizeof(header));
    wf.write(reinterpret_cast<:::no-loc(char):::*>(begin), bufferSize);

    ::::no-loc(delete):::[] buffer; // Return memory to the OS.
    wf.close();
}

```

:::no-loc(void):::* ponteiros

Um ponteiro para `:::no-loc(void):::` simplesmente aponta para um local de memória bruto. Às vezes, é necessário usar `:::no-loc(void):::*` ponteiros, por exemplo, ao passar entre o código C++ e as funções C.

Quando um ponteiro tipado é convertido em um `:::no-loc(void)>:::` ponteiro, o conteúdo do local da memória não é alterado. No entanto, as informações de tipo são perdidas, para que você não possa fazer operações de incrementar ou decrementar. Um local de memória pode ser convertido, por exemplo, de `MyClass*` para `:::no-loc(void):::*` e de volta para `MyClass*`. Essas operações são inherentemente propensas a erros e exigem muito cuidado com `:::no-loc(void)>:::` erros. O C++ moderno desencoraja o uso de `:::no-loc(void)>:::` ponteiros em quase todas as circunstâncias.

```

//func.c
:::no-loc(void)::: func(:::no-loc(void):::* data, int length)
{
    :::no-loc(char):::* c = (:::no-loc(char):::*)(data);

    // fill in the buffer with data
    for (int i = 0; i < length; ++i)
    {
        *c = 0x41;
        ++c;
    }
}

// main.cpp
#include <iostream>

extern "C"
{
    :::no-loc(void)::: func(:::no-loc(void):::* data, int length);
}

class MyClass
{
public:
    int num;
    std::string name;
    :::no-loc(void)::: print() { std::cout << name << ":" << num << std::endl; }
};

int main()
{
    MyClass* mc = :::no-loc(new)::: MyClass{10, "Marian"};
    :::no-loc(void):::* p = static_cast<:::no-loc(void):::*>(mc);
    MyClass* mc2 = static_cast<MyClass*>(p);
    std::cout << mc2->name << std::endl; // "Marian"

    // use operator :::no-loc(new)::: to allocate untyped memory block
    :::no-loc(void):::* p:::no-loc(void)::: = operator :::no-loc(new):::(1000);
    :::no-loc(char):::* p:::no-loc(char)::: = static_cast<:::no-loc(char):::*>(p:::no-loc(void):::);
    for(:::no-loc(char):::* c = p:::no-loc(char):::; c < p:::no-loc(char)::: + 1000; ++c)
    {
        *c = 0x00;
    }
    func(p:::no-loc(void):::, 1000);
    :::no-loc(char)::: ch = static_cast<:::no-loc(char):::*>(p:::no-loc(void):::)[0];
    std::cout << ch << std::endl; // 'A'
    operator :::no-loc(delete):::(p);
}

```

Ponteiros para funções

Na programação em estilo C, os ponteiros de função são usados principalmente para passar funções para outras funções. Essa técnica permite que o chamador Personalize o comportamento de uma função sem modificá-la. No C++ moderno, as [expressões lambda](#) fornecem o mesmo recurso com maior segurança de tipos e outras vantagens.

Uma declaração de ponteiro de função especifica a assinatura que a função indicada para deve ter:

```
// Declare pointer to any function that...

// ...accepts a string and returns a string
string (*g)(string a);

// has no return value and no parameters
:::no-loc(void)::: (*x)();

// ...returns an int and takes three parameters
// of the specified types
int (*i)(int i, string s, double d);
```

O exemplo a seguir mostra uma função `combine` que usa como um parâmetro qualquer função que aceita um `std::string` e retorna um `std::string`. Dependendo da função que é passada para `combine`, ela precede ou acrescenta uma cadeia de caracteres.

```
#include <iostream>
#include <string>

using namespace std;

string base {"hello world"};

string append(string s)
{
    return base.append(" ").append(s);
}

string prepend(string s)
{
    return s.append(" ").append(base);
}

string combine(string s, string(*g)(string a))
{
    return (*g)(s);
}

int main()
{
    cout << combine("from MSVC", append) << "\n";
    cout << combine("Good morning and", prepend) << "\n";
}
```

Confira também

[Ponteiros inteligentes Operador de indireção: *](#)

[Operador address-of: &](#)

[Bem-vindo de volta ao C++](#)

Ponteiros const e volatile

02/09/2020 • 6 minutes to read • [Edit Online](#)

As palavras-chave `const` e `volatile` alteram como os ponteiros são tratados. A `const` palavra-chave especifica que o ponteiro não pode ser modificado após a inicialização; o ponteiro é protegido contra modificações posteriormente.

A `volatile` palavra-chave especifica que o valor associado ao nome que se segue pode ser modificado por ações diferentes daquelas no aplicativo do usuário. Portanto, a `volatile` palavra-chave é útil para declarar objetos na memória compartilhada que pode ser acessada por vários processos ou áreas de dados globais usadas para comunicação com rotinas de serviço de interrupção.

Quando um nome é declarado como `volatile`, o compilador recarrega o valor da memória toda vez que é acessado pelo programa. Isso reduz drasticamente as otimizações possíveis. No entanto, quando o estado de um objeto pode ser alterado inesperadamente, é a única maneira de assegurar o desempenho previsível do programa.

Para declarar o objeto apontado pelo ponteiro como `const` ou `volatile`, use uma declaração do formulário:

```
const char *cpch;
volatile char *vpch;
```

Para declarar o valor do ponteiro — ou seja, o endereço real armazenado no ponteiro — como `const` ou `volatile`, use uma declaração do formulário:

```
char * const pchc;
char * volatile pchv;
```

A linguagem C++ impede atribuições que permitiriam a modificação de um objeto ou ponteiro declarado como `const`. Essas atribuições removeriam as informações com as quais o objeto ou o ponteiro foi declarado, violando assim a intenção da declaração original. Considere as seguintes declarações:

```
const char cch = 'A';
char ch = 'B';
```

Dadas as declarações anteriores de dois objetos (`cch`, do tipo `const char` e `ch`, do tipo `Char`), as seguintes declarações/inicializações são válidas:

```
const char *pch1 = &cch;
const char *const pch4 = &cch;
const char *pch5 = &ch;
char *pch6 = &ch;
char *const pch7 = &ch;
const char *const pch8 = &ch;
```

As seguintes declarações/inicializações são errôneas.

```
char *pch2 = &cch; // Error
char *const pch3 = &cch; // Error
```

A declaração de `pch2` declara um ponteiro em que um objeto constante pode ser modificado e, portanto, não é permitida. A declaração de `pch3` especifica que o ponteiro é constante, não o objeto; a declaração não é permitida pelo mesmo motivo que a `pch2` declaração não é permitida.

As oito atribuições a seguir mostram a atribuição por ponteiro e a alteração do valor do ponteiro para as declarações anteriores; por enquanto, suponha que a inicialização estava correta para `pch1` a `pch8`.

```
*pch1 = 'A'; // Error: object declared const
pch1 = &ch; // OK: pointer not declared const
*pch2 = 'A'; // OK: normal pointer
pch2 = &ch; // OK: normal pointer
*pch3 = 'A'; // OK: object not declared const
pch3 = &ch; // Error: pointer declared const
*pch4 = 'A'; // Error: object declared const
pch4 = &ch; // Error: pointer declared const
```

Ponteiros declarados como `volatile`, ou como uma combinação de `const` e `volatile`, obedecem às mesmas regras.

Ponteiros para `const` objetos geralmente são usados em declarações de função da seguinte maneira:

```
errno_t strcpy_s( char *strDestination, size_t numberOfElements, const char *strSource );
```

A instrução anterior declara uma função, `strcpy_s`, onde dois dos três argumentos são do tipo ponteiro para `char`. Como os argumentos são passados por referência e não por valor, a função seria livre para modificar `strDestination` e `strSource` se `strSource` não fosse declarada como `const`. A declaração de `strSource` como `const` garante o chamador que `strSource` não pode ser alterado pela função chamada.

NOTE

Como há uma conversão padrão de `TypeName*` para `const TypeName*`, é legal passar um argumento do tipo `char *` para `strcpy_s`. No entanto, o inverso não é verdadeiro; Não existe conversão implícita para remover o `const` atributo de um objeto ou ponteiro.

Um `const` ponteiro de um determinado tipo pode ser atribuído a um ponteiro do mesmo tipo. No entanto, um ponteiro não `const` pode ser atribuído a um `const` ponteiro. O código a seguir mostra atribuições corretas e incorretas:

```
// const_pointer.cpp
int *const cpObject = 0;
int *pObject;

int main() {
    pObject = cpObject;
    cpObject = pObject; // C3892
}
```

O exemplo a seguir mostra como declarar um objeto como `const` se você tiver um ponteiro para um ponteiro para um objeto.

```
// const_pointer2.cpp
struct X {
    X(int i) : m_i(i) { }
    int m_i;
};

int main() {
    // correct
    const X cx(10);
    const X * pcx = &cx;
    const X ** ppcx = &pcx;

    // also correct
    X const cx2(20);
    X const * pcx2 = &cx2;
    X const ** ppcx2 = &pcx2;
}
```

Confira também

[Ponteiros do Ponteiros brutos](#)

Operadores `new` e `delete`

02/09/2020 • 10 minutes to read • [Edit Online](#)

O C++ dá suporte à alocação dinâmica e à desalocação de objetos usando os `new` e `delete` operadores. Esses operadores alocam memória para objetos de um pool chamado de repositório livre. O `new` operador chama a função especial `operator new` e o `delete` operador chama a função especial `operator delete`.

A `new` função na biblioteca padrão C++ dá suporte ao comportamento especificado no padrão C++, que deve gerar uma `std::bad_alloc` exceção se a alocação de memória falhar. Se você ainda quiser a versão sem lançamento do `new`, vincule seu programa com o `nothrownew.obj`. No entanto, quando você vincula com `nothrownew.obj`, o padrão `operator new` na biblioteca C++ Standard não funciona mais.

Para obter uma lista dos arquivos de biblioteca na biblioteca de tempo de execução C e na biblioteca padrão C++, consulte [recursos da biblioteca CRT](#).

O `new` operador

O compilador traduz uma instrução como esta em uma chamada para a função `operator new`:

```
char *pch = new char[BUFFER_SIZE];
```

Se a solicitação for de zero bytes de armazenamento, o `operator new` retornará um ponteiro para um objeto distinto. Ou seja, chamadas repetidas para `operator new` retornar ponteiros diferentes. Se não houver memória suficiente para a solicitação de alocação, `operator new` o lançará uma `std::bad_alloc` exceção. Ou, ele retornará `nullptr` se você tiver vinculado ao suporte de não lançamento `operator new`.

Você pode escrever uma rotina que tente liberar memória e tentar a alocação novamente. Para obter mais informações, consulte [_set_new_handler](#). Para obter detalhes sobre o esquema de recuperação, consulte a seção [lidando com memória insuficiente](#).

Os dois escopos para `operator new` funções são descritos na tabela a seguir.

Escopo para `operator new` funções

OPERADOR	ESSCOPO
<code>::operator new</code>	Global
<code>nome da classe*** ::operator new *</code>	Classe

O primeiro argumento `operator new` deve ser do tipo `size_t`, definido em `<stddef.h>` e o tipo de retorno é sempre `void*`.

A `operator new` função global é chamada quando o `new` operador é usado para alocar objetos de tipos internos, objetos do tipo de classe que não contêm funções definidas pelo usuário `operator new` e matrizes de qualquer tipo. Quando o `new` operador é usado para alocar objetos de um tipo de classe onde um `operator new` é definido, essa classe `operator new` é chamada.

Uma `operator new` função definida para uma classe é uma função membro estática (que não pode ser virtual) que oculta a `operator new` função global para objetos desse tipo de classe. Considere o caso em que `new` é

usado para alocar e definir a memória para um determinado valor:

```
#include <malloc.h>
#include <memory.h>

class Blanks
{
public:
    Blanks(){}
    void *operator new( size_t stAllocateBlock, char chInit );
};

void *Blanks::operator new( size_t stAllocateBlock, char chInit )
{
    void *pvTemp = malloc( stAllocateBlock );
    if( pvTemp != 0 )
        memset( pvTemp, chInit, stAllocateBlock );
    return pvTemp;
}

// For discrete objects of type Blanks, the global operator new function
// is hidden. Therefore, the following code allocates an object of type
// Blanks and initializes it to 0xa5
int main()
{
    Blanks *a5 = new(0xa5) Blanks;
    return a5 != 0;
}
```

O argumento fornecido entre parênteses para `new` é passado `Blanks::operator new` como o `chInit` argumento. No entanto, a `operator new` função global está oculta, causando um código como o seguinte para gerar um erro:

```
Blanks *SomeBlanks = new Blanks;
```

O compilador oferece suporte à matriz de membros `new` e `delete` operadores em uma declaração de classe. Por exemplo:

```
class MyClass
{
public:
    void * operator new[] (size_t)
    {
        return 0;
    }
    void operator delete[] (void*)
    {
    }
};

int main()
{
    MyClass *p MyClass = new MyClass[5];
    delete [] p MyClass;
}
```

Manipulação de memória insuficiente

O teste de alocação de memória com falha pode ser feito conforme mostrado aqui:

```

#include <iostream>
using namespace std;
#define BIG_NUMBER 100000000
int main() {
    int *pI = new int[BIG_NUMBER];
    if( pI == 0x0 ) {
        cout << "Insufficient memory" << endl;
        return -1;
    }
}

```

Há outra maneira de lidar com falhas de solicitações de alocação de memória. Escreva uma rotina de recuperação personalizada para lidar com essa falha e, em seguida, registre sua função chamando a [_set_new_handler](#) função de tempo de execução.

O `delete` operador

A memória que é alocada dinamicamente usando o `new` operador pode ser liberada usando o `delete` operador. O operador `Delete` chama a `operator delete` função, que libera a memória de volta para o pool disponível. O uso do `delete` operador também faz com que o destruidor de classe (se houver) seja chamado.

Há funções globais e com escopo de classe `operator delete`. Apenas uma `operator delete` função pode ser definida para uma determinada classe; se definida, ela ocultará a `operator delete` função global. A `operator delete` função global é sempre chamada para matrizes de qualquer tipo.

A `operator delete` função global. Existem dois formulários para as `operator delete` funções global e membro de classe `operator delete`:

```

void operator delete( void * );
void operator delete( void *, size_t );

```

Somente um dos dois formulários anteriores pode estar presente para uma determinada classe. O primeiro formulário usa um único argumento do tipo `void *`, que contém um ponteiro para o objeto a ser desalocado. O segundo formulário, a desalocação dimensionada, usa dois argumentos: o primeiro é um ponteiro para o bloco de memória a ser desalocado e o segundo é o número de bytes a serem desalocados. O tipo de retorno de ambos os formulários é `void` (`operator delete` não pode retornar um valor).

A intenção do segundo formulário é acelerar a pesquisa da categoria de tamanho correto do objeto a ser excluído. Essas informações geralmente não são armazenadas perto da própria alocação e provavelmente não são armazenadas em cache. O segundo formulário é útil quando uma `operator delete` função de uma classe base é usada para excluir um objeto de uma classe derivada.

A `operator delete` função é estática, portanto, não pode ser virtual. A `operator delete` função obedece ao controle de acesso, conforme descrito em [controle de acesso de membro](#).

O exemplo a seguir mostra as funções definidas pelo usuário `operator new` e `operator delete` projetadas para alocações de log e desalocações de memória:

```

#include <iostream>
using namespace std;

int fLogMemory = 0;      // Perform logging (0=no; nonzero=yes)?
int cBlocksAllocated = 0; // Count of blocks allocated.

// User-defined operator new.
void *operator new( size_t stAllocateBlock ) {
    static int fInOpNew = 0; // Guard flag.

    if ( fLogMemory && !fInOpNew ) {
        fInOpNew = 1;
        clog << "Memory block " << ++cBlocksAllocated
            << " allocated for " << stAllocateBlock
            << " bytes\n";
        fInOpNew = 0;
    }
    return malloc( stAllocateBlock );
}

// User-defined operator delete.
void operator delete( void *pvMem ) {
    static int fInOpDelete = 0; // Guard flag.
    if ( fLogMemory && !fInOpDelete ) {
        fInOpDelete = 1;
        clog << "Memory block " << cBlocksAllocated--
            << " deallocated\n";
        fInOpDelete = 0;
    }

    free( pvMem );
}

int main( int argc, char *argv[] ) {
    fLogMemory = 1; // Turn logging on
    if( argc > 1 )
        for( int i = 0; i < atoi( argv[1] ); ++i ) {
            char *pMem = new char[10];
            delete[] pMem;
        }
    fLogMemory = 0; // Turn logging off.
    return cBlocksAllocated;
}

```

O código anterior pode ser usado para detectar "vazamento de memória", ou seja, memória alocada no armazenamento gratuito, mas nunca liberada. Para detectar vazamentos, os `new` operadores globais e `delete` são redefinidos para contar a alocação e a desalocação da memória.

O compilador oferece suporte à matriz de membros `new` e `delete` operadores em uma declaração de classe. Por exemplo:

```
// spec1_the_operator_delete_function2.cpp
// compile with: /c
class X {
public:
    void * operator new[] (size_t) {
        return 0;
    }
    void operator delete[] (void*) {}
};

void f() {
    X *pX = new X[5];
    delete [] pX;
}
```

Ponteiros inteligentes (C++ moderno)

02/09/2020 • 15 minutes to read • [Edit Online](#)

Na programação C++ moderna, a biblioteca padrão inclui *ponteiros inteligentes*, que são usados para ajudar a garantir que os programas estejam livres de vazamentos de memória e de recursos e sejam seguros para exceções.

Uso de ponteiros inteligentes

Os ponteiros inteligentes são definidos no `std` namespace no `<memory>` arquivo de cabeçalho. Eles são cruciais para a [RAII](#) ou a *aquisição de recursos* é a linguagem de programação de inicialização. O objetivo principal dessa linguagem é garantir que a aquisição de recursos ocorra ao mesmo tempo em que o objeto é inicializado, de forma que todos os recursos do objeto sejam criados e preparados em uma linha de código. Em termos práticos, o princípio fundamental da linguagem RAII é fornecer a propriedade de qualquer recurso alocado a heap, por exemplo, memória alocada dinamicamente ou identificadores de objetos do sistema, a um objeto alocado em pilha cujo destruidor contenha o código para excluir ou liberar o recurso e também qualquer código de limpeza associado.

Na maioria dos casos, quando você inicializa um ponteiro bruto ou identificador de recursos para apontar para um recurso real, transforma o ponteiro em ponteiro inteligente imediatamente. Em C++ moderno, os ponteiros brutos são usados somente em pequenos blocos de código de escopo limitado, loops ou funções auxiliares onde o desempenho é essencial e não há possibilidade de confusão sobre a propriedade.

O exemplo a seguir compara uma declaração de ponteiro bruto a uma declaração de ponteiro inteligente.

```
void UseRawPointer()
{
    // Using a raw pointer -- not recommended.
    Song* pSong = new Song(L"Nothing on You", L"Bruno Mars");

    // Use pSong...

    // Don't forget to delete!
    delete pSong;
}

void UseSmartPointer()
{
    // Declare a smart pointer on stack and pass it the raw pointer.
    unique_ptr<Song> song2(new Song(L"Nothing on You", L"Bruno Mars"));

    // Use song2...
    wstring s = song2->duration_;
    //...

} // song2 is deleted automatically here.
```

Conforme mostrado no exemplo, um ponteiro inteligente é um modelo de classe que você declara na pilha e inicializa usando um ponteiro bruto que aponta para um objeto alocado a heap. Depois que o ponteiro inteligente é inicializado, ele possui o ponteiro bruto. Isso significa que o ponteiro inteligente é responsável pela exclusão da memória especificada pelo ponteiro bruto. O destruidor do ponteiro inteligente contém a chamada para exclusão e, como o ponteiro inteligente é declarado na pilha, seu destruidor é chamado quando o ponteiro inteligente fica fora do escopo, mesmo se uma exceção for lançada posteriormente na pilha.

Acesse o ponteiro encapsulado usando os operadores de ponteiros familiares, `->` e `*`, que a classe do ponteiro inteligente sobrecarrega para retornar o ponteiro bruto encapsulado.

A linguagem de ponteiro inteligente C++ é semelhante à criação de objeto em linguagens como C#: você cria o objeto e permite que o sistema cuide de sua exclusão no momento certo. A diferença é que nenhum coletor de lixo separado é executado em segundo plano; a memória é gerenciada com as regras de escopo C++ padrão de modo que o ambiente em runtime seja mais rápido e mais eficiente.

IMPORTANT

Crie sempre ponteiros inteligentes em uma linha de código separada, nunca em uma lista de parâmetros, de forma que um vazamento sutil de recursos não ocorre devido a determinadas regras de alocação da lista de parâmetros.

O exemplo a seguir mostra como um `unique_ptr` tipo de ponteiro inteligente da biblioteca padrão C++ pode ser usado para encapsular um ponteiro para um objeto grande.

```
class LargeObject
{
public:
    void DoSomething(){}
};

void ProcessLargeObject(const LargeObject& lo){}
void SmartPointerDemo()
{
    // Create the object and pass it to a smart pointer
    std::unique_ptr<LargeObject> pLarge(new LargeObject());

    //Call a method on the object
    pLarge->DoSomething();

    // Pass a reference to a method.
    ProcessLargeObject(*pLarge);

} //pLarge is deleted automatically when function block goes out of scope.
```

O exemplo demonstra as etapas essenciais a seguir para o uso de ponteiros inteligentes.

1. Declare o ponteiro inteligente como uma variável automática (local). (Não use a `new` expressão ou `malloc` no próprio ponteiro inteligente.)
2. No parâmetro de tipo, especifique o tipo apontado do ponteiro encapsulado.
3. Passe um ponteiro bruto para um `new` objeto-Ed no construtor de ponteiro inteligente. (Algumas funções do utilitário ou construtores de ponteiro inteligente fazem isso para você.)
4. Use os operadores `->` e `*` sobrecarregados para acessar o objeto.
5. Deixe o ponteiro inteligente excluir o objeto.

Ponteiros inteligentes são criados para terem a maior eficiência possível em termos de memória e de desempenho. Por exemplo, o único membro de dados em `unique_ptr` é o ponteiro encapsulado. Isso significa que `unique_ptr` é exatamente do mesmo tamanho que o ponteiro, com quatro bytes ou com oito bytes. O acesso ao ponteiro encapsulado usando os operadores de ponteiro inteligente sobrecarregado `* e->` não é significativamente mais lento do que acessar os ponteiros brutos diretamente.

Os ponteiros inteligentes têm suas próprias funções de membro, que são acessadas usando a notação "ponto". Por exemplo, alguns ponteiros inteligentes da biblioteca padrão C++ têm uma função de membro `reset` que

libera a propriedade do ponteiro. Isso é útil quando você deseja liberar a memória possuída pelo ponteiro inteligente antes que o ponteiro inteligente saia do escopo, como mostrado no exemplo a seguir.

```
void SmartPointerDemo2()
{
    // Create the object and pass it to a smart pointer
    std::unique_ptr<LargeObject> pLarge(new LargeObject());

    // Call a method on the object
    pLarge->DoSomething();

    // Free the memory before we exit function block.
    pLarge.reset();

    // Do some other work...
}
```

Os ponteiros inteligentes geralmente oferecem uma maneira de acessar diretamente seu ponteiro bruto. Os ponteiros inteligentes da biblioteca padrão C++ têm uma `get` função membro para essa finalidade e `ccomPtr` têm um `p` membro de classe pública. Fornecendo acesso direto ao ponteiro subjacente, você pode usar o ponteiro inteligente para gerenciar a memória em seu próprio código e ainda passar o ponteiro bruto para o código que não oferece suporte a ponteiros inteligentes.

```
void SmartPointerDemo4()
{
    // Create the object and pass it to a smart pointer
    std::unique_ptr<LargeObject> pLarge(new LargeObject());

    // Call a method on the object
    pLarge->DoSomething();

    // Pass raw pointer to a legacy API
    LegacyLargeObjectFunction(pLarge.get());
}
```

Tipos de ponteiros inteligentes

A seção a seguir resume os diferentes tipos de ponteiros inteligentes que estão disponíveis no ambiente de programação do Windows e descreve quando usá-los.

Ponteiros inteligentes da biblioteca padrão C++

Use esses ponteiros inteligentes como primeira opção para o encapsulamento de ponteiros para objetos C++ antigos simples (POCO).

- `unique_ptr`

Permite exatamente um proprietário do ponteiro subjacente. Use como a opção padrão para POCO, a menos que você tenha certeza de que precisa de um `shared_ptr`. Pode ser movido para um novo proprietário, mas não copiado ou compartilhado. Substitui `auto_ptr`, que será preferido. Compare com `boost::scoped_ptr`. `unique_ptr` é pequeno e eficiente; o tamanho é um ponteiro e dá suporte a referências rvalue para inserção e recuperação rápidas de coleções de bibliotecas padrão do C++.

Arquivo de cabeçalho: `<memory>`. Para obter mais informações, consulte [como: criar e usar instâncias de `unique_ptr` e `unique_ptr` classe](#).

- `shared_ptr`

Ponteiro inteligente contado por referência. Use quando quiser atribuir um ponteiro bruto a vários proprietários, por exemplo, ao retornar uma cópia de um ponteiro de um contêiner, porém mantendo o

original. O ponteiro bruto não será excluído até que todos os proprietários de `shared_ptr` tenham saído do escopo ou tenham desistido da propriedade. O tamanho é de dois ponteiros; um para o objeto e um para o bloco de controle compartilhado que contém a contagem de referência. Arquivo de cabeçalho: `<memory>`. Para obter mais informações, consulte [como: criar e usar instâncias de shared_ptr e shared_ptr classe](#).

- `weak_ptr`

Ponteiro inteligente de casos especiais para uso em conjunto com `shared_ptr`. Um `weak_ptr` fornece acesso a um objeto pertencente a uma ou mais instâncias de `shared_ptr`, mas não participa da contagem de referência. Use quando você quiser observar um objeto, mas sem exigir que ele permaneça ativo. Necessário em alguns casos para interromper referências circulares entre instâncias `shared_ptr`. Arquivo de cabeçalho: `<memory>`. Para obter mais informações, consulte [como: criar e usar instâncias de weak_ptr e weak_ptr classe](#).

Ponteiros inteligentes para objetos COM (programação clássica do Windows)

Ao trabalhar com objetos COM, coloque os ponteiros de interface em um tipo de ponteiro inteligente apropriado. A Biblioteca de Modelos Ativos (ATL) define vários ponteiros inteligentes para várias finalidades. Você também pode usar o tipo de ponteiro inteligente `_com_ptr_t`, que o compilador usa ao criar classes wrapper dos arquivos .tlb. É a melhor opção quando você não quer incluir os arquivos de cabeçalho da ATL.

Classe `CComPtr`

Use isso a menos que você não possa usar a ATL. Executa a contagem de referência usando os métodos `AddRef` e `Release`. Para obter mais informações, consulte [como: criar e usar instâncias de CComPtr e CComQIPtr](#).

Classe `CComQIPtr`

É semelhante a `CComPtr` mas também fornece a sintaxe simplificada para chamar `QueryInterface` em objetos COM. Para obter mais informações, consulte [como: criar e usar instâncias de CComPtr e CComQIPtr](#).

Classe `CComHeapPtr`

Ponteiro inteligente para objetos que usam `CoTaskMemFree` para liberar memória.

Classe `CComGITPtr`

Ponteiro inteligente para as interfaces que são obtidas da tabela de interface global (GIT).

Classe `_com_ptr_t`

É semelhante a `CComQIPtr` em funcionalidade, mas não depende de cabeçalhos da ATL.

Ponteiros inteligentes ATL para objetos POCO

Além de ponteiros inteligentes para objetos COM, a ATL também define os ponteiros inteligentes e as coleções de ponteiros inteligentes, para POCO (objetos C++ antigos). Na programação clássica do Windows, esses tipos são alternativas úteis para as coleções de bibliotecas padrão do C++, especialmente quando a portabilidade do código não é necessária ou quando você não deseja misturar os modelos de programação da biblioteca C++ Standard e da ATL.

Classe `CAutoPtr`

Ponteiro inteligente que impõe a propriedade exclusiva transferindo a propriedade na cópia. Comparável à classe `std::auto_ptr` preterida.

Classe `CHheapPtr`

Ponteiro inteligente para objetos que são alocados usando a função de `malloc` C.

Classe `CAutoVectorPtr`

Ponteiro inteligente para matrizes que são alocadas usando `new[]`.

Classe `CAutoPtrArray`

Classe que encapsula uma matriz de elementos [CAutoPtr](#) .

[Classe CAutoPtrList](#)

Classe que encapsula métodos para manipular uma lista de nós [CAutoPtr](#) .

Confira também

[Ponteiros](#)

[Referência da linguagem C++](#)

[Biblioteca padrão do C++](#)

Como criar e usar instâncias de unique_ptr

02/12/2019 • 4 minutes to read • [Edit Online](#)

Um `unique_ptr` não compartilha seu ponteiro. Ele não pode ser copiado para outro `unique_ptr`, passado pelo valor para uma função ou usado em C++ qualquer algoritmo de biblioteca padrão que exija a tomada de cópias. Um `unique_ptr` só pode ser movido. Isso significa que a propriedade do recurso de memória é transferida para outro `unique_ptr` e que o `unique_ptr` original não a possui mais. É recomendável que você restrinja um objeto a um proprietário, porque a propriedade múltipla adiciona complexidade à lógica do programa. Portanto, quando você precisar de um ponteiro inteligente para um C++ objeto simples, use `unique_ptr` e, ao construir um `unique_ptr`, use a função auxiliar `make_unique`.

O diagrama a seguir ilustra a transferência de propriedade entre duas instâncias de `unique_ptr`.

```
auto ptrA = make_unique<Song>(L"Diana Krall", L"The Look of Love");
```



```
auto ptrB = std::move(ptrA);
```



`unique_ptr` é definido no cabeçalho `<memory>` na biblioteca C++ padrão. Ele é exatamente tão eficiente quanto um ponteiro bruto e pode ser usado em C++ contêineres de biblioteca padrão. A adição de instâncias de `unique_ptr` C++ para contêineres de biblioteca padrão é eficiente porque o construtor de movimentação do `unique_ptr` elimina a necessidade de uma operação de cópia.

Exemplo 1

O exemplo a seguir mostra como criar instâncias de `unique_ptr` e passá-las entre as funções.

```
unique_ptr<Song> SongFactory(const std::wstring& artist, const std::wstring& title)
{
    // Implicit move operation into the variable that stores the result.
    return make_unique<Song>(artist, title);
}

void MakeSongs()
{
    // Create a new unique_ptr with a new object.
    auto song = make_unique<Song>(L"Mr. Children", L"Namonaki Uta");

    // Use the unique_ptr.
    vector<wstring> titles = { song->title };

    // Move raw pointer from one unique_ptr to another.
    unique_ptr<Song> song2 = std::move(song);

    // Obtain unique_ptr from function that returns by value.
    auto song3 = SongFactory(L"Michael Jackson", L"Beat It");
}
```

Esses exemplos demonstram esta característica básica de `unique_ptr`: pode ser movido, mas não copiado. "Mover" transfere a propriedade a um novo `unique_ptr` e redefine o `unique_ptr` antigo.

Exemplo 2

O exemplo a seguir mostra como criar instâncias de `unique_ptr` e usá-las em um vetor.

```
void SongVector()
{
    vector<unique_ptr<Song>> songs;

    // Create a few new unique_ptr<Song> instances
    // and add them to vector using implicit move semantics.
    songs.push_back(make_unique<Song>(L"B'z", L"Juice"));
    songs.push_back(make_unique<Song>(L"Namie Amuro", L"Funky Town"));
    songs.push_back(make_unique<Song>(L"Kome Kome Club", L"Kimi ga Iru Dake de"));
    songs.push_back(make_unique<Song>(L"Ayumi Hamasaki", L"Poker Face"));

    // Pass by const reference when possible to avoid copying.
    for (const auto& song : songs)
    {
        wcout << L"Artist: " << song->artist << L"    Title: " << song->title << endl;
    }
}
```

No intervalo do loop, observe que `unique_ptr` é passado por referência. Se você tentar passar por valor aqui, o compilador lançará um erro porque o construtor de cópia `unique_ptr` foi excluído.

Exemplo 3

O exemplo a seguir mostra como inicializar um `unique_ptr` que é membro da classe.

```
class MyClass
{
private:
    // MyClass owns the unique_ptr.
    unique_ptr<ClassFactory> factory;
public:

    // Initialize by using make_unique with ClassFactory default constructor.
    MyClass() : factory ( make_unique<ClassFactory>())
    {}

    void MakeClass()
    {
        factory->DoSomething();
    }
};
```

Exemplo 4

Você pode usar `make_unique` para criar um `unique_ptr` a uma matriz, mas não pode usar `make_unique` para inicializar os elementos da matriz.

```
// Create a unique_ptr to an array of 5 integers.
auto p = make_unique<int[]>(5);

// Initialize the array.
for (int i = 0; i < 5; ++i)
{
    p[i] = i;
    wcout << p[i] << endl;
}
```

Para obter mais exemplos, consulte [make_unique](#).

Consulte também

[Ponteiros inteligentes \(C++ moderno\)](#)

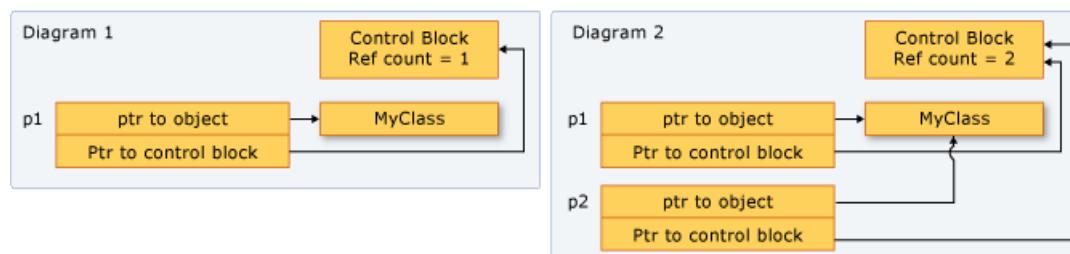
[make_unique](#)

Como criar e usar instâncias de shared_ptr

02/09/2020 • 11 minutes to read • [Edit Online](#)

O tipo `shared_ptr` é um ponteiro inteligente na biblioteca padrão do C++ que foi projetado para cenários em que mais de um proprietário pode ter que gerenciar o tempo de vida do objeto na memória. Depois de inicializar um `shared_ptr`, você pode copiá-lo, passá-lo pelo valor em argumentos de função e atribuí-lo a outras instâncias `shared_ptr`. Todas as instâncias apontam para o mesmo objeto e compartilham o acesso a um "bloco de controle" que incrementa e decrementa a contagem de referência sempre que um novo `shared_ptr` é adicionado, sai do escopo ou é redefinido. Quando a contagem de referência alcança zero, o bloco de controle exclui o recurso de memória e ele mesmo.

A ilustração a seguir mostra várias instâncias `shared_ptr` que apontam para um local da memória.



Configuração de exemplo

Os exemplos a seguir supõem que você incluiu os cabeçalhos necessários e declarou os tipos necessários, conforme mostrado aqui:

```

// shared_ptr-examples.cpp
// The following examples assume these declarations:
#include <algorithm>
#include <iostream>
#include <memory>
#include <string>
#include <vector>

struct MediaAsset
{
    virtual ~MediaAsset() = default; // make it polymorphic
};

struct Song : public MediaAsset
{
    std::wstring artist;
    std::wstring title;
    Song(const std::wstring& artist_, const std::wstring& title_) :
        artist{ artist_ }, title{ title_ } {}
};

struct Photo : public MediaAsset
{
    std::wstring date;
    std::wstring location;
    std::wstring subject;
    Photo(
        const std::wstring& date_,
        const std::wstring& location_,
        const std::wstring& subject_) :
        date{ date_ }, location{ location_ }, subject{ subject_ } {}
};

using namespace std;

int main()
{
    // The examples go here, in order:
    // Example 1
    // Example 2
    // Example 3
    // Example 4
    // Example 6
}

```

Exemplo 1

Sempre que possível, use a função `make_shared` para criar um `shared_ptr` quando o recurso de memória for criado pela primeira vez. `make_shared` é à prova de exceção. Ele usa a mesma chamada para alocar a memória para o bloco de controle e o recurso, que reduz a sobrecarga de construção. Se você não usar `make_shared`, você precisará usar uma expressão explícita `new` para criar o objeto antes de passá-lo para o `shared_ptr` Construtor. O exemplo a seguir mostra várias maneiras de declarar e inicializar um `shared_ptr` junto com um novo objeto.

```

// Use make_shared function when possible.
auto sp1 = make_shared<Song>(L"The Beatles", L"Im Happy Just to Dance With You");

// Ok, but slightly less efficient.
// Note: Using new expression as constructor argument
// creates no named variable for other code to access.
shared_ptr<Song> sp2(new Song(L"Lady Gaga", L"Just Dance"));

// When initialization must be separate from declaration, e.g. class members,
// initialize with nullptr to make your programming intent explicit.
shared_ptr<Song> sp5(nullptr);
//Equivalent to: shared_ptr<Song> sp5;
//...
sp5 = make_shared<Song>(L"Elton John", L"I'm Still Standing");

```

Exemplo 2

O exemplo a seguir mostra como declarar e inicializar instâncias `shared_ptr` que assumem a posse compartilhada de um objeto que já foi alocado por outro `shared_ptr`. Suponha que `sp2` seja um `shared_ptr` inicializado.

```

//Initialize with copy constructor. Increments ref count.
auto sp3(sp2);

//Initialize via assignment. Increments ref count.
auto sp4 = sp2;

//Initialize with nullptr. sp7 is empty.
shared_ptr<Song> sp7(nullptr);

// Initialize with another shared_ptr. sp1 and sp2
// swap pointers as well as ref counts.
sp1.swap(sp2);

```

Exemplo 3

`shared_ptr` também será útil em contêineres da Biblioteca Padrão do C++ quando você estiver usando algoritmos que copiam elementos. Você pode encapsular os elementos em um `shared_ptr` e, em seguida, copiá-lo em outros contêineres sabendo que a memória subjacente é válida somente pelo tempo em que for necessária. O exemplo a seguir mostra como usar o algoritmo `remove_copy_if` em instâncias `shared_ptr` em um vetor.

```

vector<shared_ptr<Song>> v;

v.push_back(make_shared<Song>(L"Bob Dylan", L"The Times They Are A Changing"));
v.push_back(make_shared<Song>(L"Aretha Franklin", L"Bridge Over Troubled Water"));
v.push_back(make_shared<Song>(L"Thalia", L"Entre El Mar y Una Estrella"));

vector<shared_ptr<Song>> v2;
remove_copy_if(v.begin(), v.end(), back_inserter(v2), [] (shared_ptr<Song> s)
{
    return s->artist.compare(L"Bob Dylan") == 0;
});

for (const auto& s : v2)
{
    wcout << s->artist << L":" << s->title << endl;
}

```

Exemplo 4

Você pode usar `dynamic_pointer_cast`, `static_pointer_cast` e `const_pointer_cast` para converter um `shared_ptr`. Essas funções são semelhantes aos `dynamic_cast`, `static_cast` operadores, e `const_cast`. O exemplo a seguir mostra como testar o tipo derivado de cada elemento em um vetor de `shared_ptr` de classes base e, em seguida, copiar os elementos e exibir informações sobre eles.

```
vector<shared_ptr<MediaAsset>> assets;

assets.push_back(shared_ptr<Song>(new Song(L"Himesh Reshammiya", L"Tera Surroor")));
assets.push_back(shared_ptr<Song>(new Song(L"Penaz Masani", L"Tu Dil De De")));
assets.push_back(shared_ptr<Photo>(new Photo(L"2011-04-06", L"Redmond, WA", L"Soccer field at Microsoft.")));

vector<shared_ptr<MediaAsset>> photos;

copy_if(assets.begin(), assets.end(), back_inserter(photos), [] (shared_ptr<MediaAsset> p) -> bool
{
    // Use dynamic_pointer_cast to test whether
    // element is a shared_ptr<Photo>.
    shared_ptr<Photo> temp = dynamic_pointer_cast<Photo>(p);
    return temp.get() != nullptr;
});

for (const auto& p : photos)
{
    // We know that the photos vector contains only
    // shared_ptr<Photo> objects, so use static_cast.
    wcout << "Photo location: " << (static_pointer_cast<Photo>(p))->location_ << endl;
}
```

Exemplo 5

Você pode passar um `shared_ptr` para outra função das seguintes maneiras:

- Passar o `shared_ptr` por valor. Isso invoca o construtor de cópia, incrementa a contagem de referência e torna o receptor um proprietário. Há uma pequena quantidade de sobrecarga nessa operação, que pode ser significativa dependendo de quantos objetos `shared_ptr` você está passando. Use essa opção quando o contrato de código implícito ou explícito entre o chamador e o receptor exigir que o receptor seja um proprietário.
- Passar o `shared_ptr` por referência ou referência const. Nesse caso, a contagem de referência não é incrementada e o receptor pode acessar o ponteiro desde que o chamador não saia do escopo. Ou o receptor pode optar por criar um `shared_ptr` com base na referência e se tornar um proprietário compartilhado. Use esta opção quando o chamador não tiver conhecimento do receptor ou quando você precisar passar um `shared_ptr` e quiser evitar a operação de cópia por motivos de desempenho.
- Passar o ponteiro subjacente ou uma referência ao objeto subjacente. Isso permite que o receptor use o objeto, mas não o habilita a compartilhar a propriedade ou estender o tempo de vida. Se o receptor criar um `shared_ptr` do ponteiro bruto, o novo `shared_ptr` será independente do original e não controlará o recurso subjacente. Use essa opção quando o contrato entre o chamador e o receptor claramente especificar que o chamador retém a propriedade do tempo de vida `shared_ptr`.
- Ao decidir como passar um `shared_ptr`, determine se o receptor deve compartilhar a propriedade do recurso subjacente. Um "proprietário" é um objeto ou uma função que pode manter o recurso subjacente ativo pelo tempo necessário. Se o chamador precisar garantir que o receptor pode estender o tempo do ponteiro para além do seu tempo de vida (da função), use a primeira opção. Se não se importar se o receptor estende o tempo de vida, passe por referência e permita ou não que o receptor o copie.

- Se tiver que fornecer um acesso de função auxiliar para o ponteiro subjacente e souber que a função auxiliar apenas usará o ponteiro e retornará antes de a função de chamada retornar, então essa função não precisará compartilhar a propriedade do ponteiro subjacente. Ela precisa apenas acessar o ponteiro dentro do tempo de vida do `shared_ptr` do chamador. Nesse caso, é seguro passar o `shared_ptr` por referência ou passar o ponteiro bruto ou uma referência ao objeto subjacente. Passar dessa maneira fornece um pequeno benefício de desempenho e também pode ajudar a expressar sua intenção de programação.
- Às vezes, por exemplo em um `std::vector<shared_ptr<T>>`, você poderá precisar passar cada `shared_ptr` a um corpo da expressão lambda ou a um objeto de função. Se o lambda ou a função não armazenar o ponteiro, passe `shared_ptr` por referência para evitar invocar o construtor de cópia para cada elemento.

Exemplo 6

O exemplo a seguir mostra como `shared_ptr` sobrecarrega vários operadores de comparação para habilitar comparações do ponteiro na memória que pertence às instâncias `shared_ptr`.

```
// Initialize two separate raw pointers.
// Note that they contain the same values.
auto song1 = new Song(L"Village People", L"YMCA");
auto song2 = new Song(L"Village People", L"YMCA");

// Create two unrelated shared_ptrs.
shared_ptr<Song> p1(song1);
shared_ptr<Song> p2(song2);

// Unrelated shared_ptrs are never equal.
wcout << "p1 < p2 = " << std::boolalpha << (p1 < p2) << endl;
wcout << "p1 == p2 = " << std::boolalpha << (p1 == p2) << endl;

// Related shared_ptr instances are always equal.
shared_ptr<Song> p3(p2);
wcout << "p3 == p2 = " << std::boolalpha << (p3 == p2) << endl;
```

Confira também

[Ponteiros inteligentes \(C++ moderno\)](#)

Como: criar e usar instâncias `weak_ptr`

02/09/2020 • 5 minutes to read • [Edit Online](#)

Às vezes, um objeto deve armazenar uma maneira de acessar o objeto subjacente de um `shared_ptr` sem fazer com que a contagem de referência seja incrementada. Normalmente, essa situação ocorre quando você tem referências cíclicas entre `shared_ptr` instâncias.

O melhor design é evitar a propriedade compartilhada de ponteiros sempre que possível. No entanto, se você precisar ter Propriedade compartilhada de `shared_ptr` instâncias, evite referências cíclicas entre elas. Quando as referências cíclicas são inevitáveis ou até mesmo preferíveis por algum motivo, use `weak_ptr` para dar a um ou mais proprietários uma referência fraca a outra `shared_ptr`. Usando um `weak_ptr`, você pode criar um `shared_ptr` que une a um conjunto existente de instâncias relacionadas, mas somente se o recurso de memória subjacente ainda for válido. Uma `weak_ptr` em si não participa da contagem de referência e, portanto, não pode impedir que a contagem de referência vá para zero. No entanto, você pode usar um `weak_ptr` para tentar obter uma nova cópia do `shared_ptr` com a qual ela foi inicializada. Se a memória já tiver sido excluída, o `weak_ptr` operador `bool` do será retornado `false`. Se a memória ainda for válida, o novo ponteiro compartilhado incrementará a contagem de referência e garantirá que a memória será válida, desde que a `shared_ptr` variável permaneça no escopo.

Exemplo

O exemplo de código a seguir mostra um caso em que `weak_ptr` é usado para garantir a exclusão adequada de objetos que têm dependências circulares. Ao examinar o exemplo, suponha que ele foi criado somente depois que as soluções alternativas foram consideradas. Os `Controller` objetos representam algum aspecto de um processo de máquina e operam de forma independente. Cada controlador deve ser capaz de consultar o status dos outros controladores a qualquer momento, e cada um deles contém um privado `vector<weak_ptr<Controller>>` para essa finalidade. Cada vetor contém uma referência circular e, portanto, as `weak_ptr` instâncias são usadas em vez de `shared_ptr`.

```
#include <iostream>
#include <memory>
#include <string>
#include <vector>
#include <algorithm>

using namespace std;

class Controller
{
public:
    int Num;
    wstring Status;
    vector<weak_ptr<Controller>> others;
    explicit Controller(int i) : Num(i), Status(L"On")
    {
        wcout << L"Creating Controller" << Num << endl;
    }

    ~Controller()
    {
        wcout << L"Destroying Controller" << Num << endl;
    }

    // Demonstrates how to test whether the
    // pointed-to memory still exists or not.
}
```

```

// Prints out memory usage of each.
void CheckStatuses() const
{
    for_each(others.begin(), others.end(), [](weak_ptr<Controller> wp) {
        auto p = wp.lock();
        if (p)
        {
            wcout << L"Status of " << p->Num << " = " << p->Status << endl;
        }
        else
        {
            wcout << L"Null object" << endl;
        }
    });
}

void RunTest()
{
    vector<shared_ptr<Controller>> v{
        make_shared<Controller>(0),
        make_shared<Controller>(1),
        make_shared<Controller>(2),
        make_shared<Controller>(3),
        make_shared<Controller>(4),
    };

    // Each controller depends on all others not being deleted.
    // Give each controller a pointer to all the others.
    for (int i = 0; i < v.size(); ++i)
    {
        for_each(v.begin(), v.end(), [&v, i](shared_ptr<Controller> p) {
            if (p->Num != i)
            {
                v[i]->others.push_back(weak_ptr<Controller>(p));
                wcout << L"push_back to v[" << i << "]: " << p->Num << endl;
            }
        });
    }

    for_each(v.begin(), v.end(), [](shared_ptr<Controller> &p) {
        wcout << L"use_count = " << p.use_count() << endl;
        p->CheckStatuses();
    });
}

int main()
{
    RunTest();
    wcout << L"Press any key" << endl;
    char ch;
    cin.getline(&ch, 1);
}

```

```
Creating Controller0
Creating Controller1
Creating Controller2
Creating Controller3
Creating Controller4
push_back to v[0]: 1
push_back to v[0]: 2
push_back to v[0]: 3
push_back to v[0]: 4
push_back to v[1]: 0
push_back to v[1]: 2
push_back to v[1]: 3
push_back to v[1]: 4
push_back to v[2]: 0
push_back to v[2]: 1
push_back to v[2]: 3
push_back to v[2]: 4
push_back to v[3]: 0
push_back to v[3]: 1
push_back to v[3]: 2
push_back to v[3]: 4
push_back to v[4]: 0
push_back to v[4]: 1
push_back to v[4]: 2
push_back to v[4]: 3
use_count = 1
Status of 1 = On
Status of 2 = On
Status of 3 = On
Status of 4 = On
use_count = 1
Status of 0 = On
Status of 2 = On
Status of 3 = On
Status of 4 = On
use_count = 1
Status of 0 = On
Status of 1 = On
Status of 3 = On
Status of 4 = On
use_count = 1
Status of 0 = On
Status of 1 = On
Status of 2 = On
Status of 4 = On
use_count = 1
Status of 0 = On
Status of 1 = On
Status of 2 = On
Status of 3 = On
Destroying Controller0
Destroying Controller1
Destroying Controller2
Destroying Controller3
Destroying Controller4
Press any key
```

Como um experimento, modifique o vetor `others` para ser um `vector<shared_ptr<Controller>>` e, em seguida, na saída, observe que nenhum destruidor é invocado quando `TestRun` retorna.

Confira também

[Ponteiros inteligentes \(C++ moderno\)](#)

Como: criar e usar instâncias CComPtr e CComQIPtr

02/12/2019 • 5 minutes to read • [Edit Online](#)

Na programação clássica do Windows, as bibliotecas geralmente são implementadas como objetos COM (ou mais precisamente, como servidores COM). Muitos componentes do sistema operacional Windows são implementados como servidores COM, e muitos colaboradores fornecem bibliotecas neste formulário. Para obter informações sobre os fundamentos do COM, consulte [Component Object Model \(com\)](#).

Quando você cria uma instância de um objeto de Component Object Model (COM), armazena o ponteiro de interface em um ponteiro inteligente COM, que executa a contagem de referência usando chamadas para `AddRef` e `Release` no destruidor. Se você estiver usando o Active Template Library (ATL) ou o biblioteca MFC (MFC), use o ponteiro inteligente do `CCComPtr`. Se você não estiver usando o ATL ou o MFC, use `com_ptr_t`. Como não há equivalente COM para `std::unique_ptr`, use esses ponteiros inteligentes para cenários de um único proprietário e de vários proprietários. `CCComPtr` e `ComQIPtr` oferecem suporte a operações de movimentação que têm referências a rvalue.

Exemplo

O exemplo a seguir mostra como usar `CCComPtr` para instanciar um objeto COM e obter ponteiros para suas interfaces. Observe que a função membro `CCComPtr::CoCreateInstance` é usada para criar o objeto COM, em vez da função Win32 que tem o mesmo nome.

```

void CComPtrDemo()
{
    HRESULT hr = CoInitialize(NULL);

    // Declare the smart pointer.
    CComPtr<IGraphBuilder> pGraph;

    // Use its member function CoCreateInstance to
    // create the COM object and obtain the IGraphBuilder pointer.
    hr = pGraph.CoCreateInstance(CLSID_FilterGraph);
    if(FAILED(hr)){ /*... handle hr error*/ }

    // Use the overloaded -> operator to call the interface methods.
    hr = pGraph->RenderFile(L"C:\\\\Users\\\\Public\\\\Music\\\\Sample Music\\\\Sleep Away.mp3", NULL);
    if(FAILED(hr)){ /*... handle hr error*/ }

    // Declare a second smart pointer and use it to
    // obtain another interface from the object.
    CComPtr<IMediaControl> pControl;
    hr = pGraph->QueryInterface(IID_PPV_ARGS(&pControl));
    if(FAILED(hr)){ /*... handle hr error*/ }

    // Obtain a third interface.
    CComPtr<IMediaEvent> pEvent;
    hr = pGraph->QueryInterface(IID_PPV_ARGS(&pEvent));
    if(FAILED(hr)){ /*... handle hr error*/ }

    // Use the second interface.
    hr = pControl->Run();
    if(FAILED(hr)){ /*... handle hr error*/ }

    // Use the third interface.
    long evCode = 0;
    pEvent->WaitForCompletion(INFINITE, &evCode);

    CoUninitialize();

    // Let the smart pointers do all reference counting.
}

```

`CComPtr` e seus parentes fazem parte da ATL e são definidos em `<atlcomcli.h>`. `_com_ptr_t` é declarado em `<comip.h>`. O compilador cria especializações de `_com_ptr_t` quando ele gera classes de wrapper para bibliotecas de tipos.

Exemplo

A ATL também fornece `CCoMQIPtr`, que tem uma sintaxe mais simples para consultar um objeto COM para recuperar uma interface adicional. No entanto, é recomendável `CComPtr` porque faz tudo o que `CCoMQIPtr` pode fazer e é semanticamente mais consistente com ponteiros de interface COM brutos. Se você usar um `CComPtr` para consultar uma interface, o novo ponteiro de interface será colocado em um parâmetro `out`. Se a chamada falhar, um `HRESULT` será retornado, que é o padrão COM típico. Com `CCoMQIPtr`, o valor de retorno é o próprio ponteiro `e`, se a chamada falhar, o valor de retorno `HRESULT` interno não poderá ser acessado. As duas linhas a seguir mostram como os mecanismos de tratamento de erros no `CComPtr` e `CCoMQIPtr` são diferentes.

```

// CComPtr with error handling:
CComPtr<IMediaControl> pControl;
hr = pGraph->QueryInterface(IID_PPV_ARGS(&pControl));
if(FAILED(hr)){ /*... handle hr error*/ }

// CComQIPtr with error handling
CComQIPtr<IMediaEvent> pEvent = pControl;
if(!pEvent){ /*... handle NULL pointer error*/ }

// Use the second interface.
hr = pControl->Run();
if(FAILED(hr)){ /*... handle hr error*/ }

```

Exemplo

`CComPtr` fornece uma especialização para `IDispatch` que permite armazenar ponteiros para componentes de automação COM e invocar os métodos na interface usando a associação tardia. `CComDispatchDriver` é um `typedef` para `CComQIPtr<IDispatch, &IID_IDispatch>`, que é implicitamente conversível para `CComPtr<IDispatch>`. Portanto, quando qualquer um desses três nomes aparece no código, ele é equivalente a `CComPtr<IDispatch>`. O exemplo a seguir mostra como obter um ponteiro para o modelo de objeto do Microsoft Word usando um `CComPtr<IDispatch>`.

```

void COMAutomationSmartPointerDemo()
{
    CComPtr<IDispatch> pWord;
    CComQIPtr<IDispatch, &IID_IDispatch> pqi = pWord;
    CComDispatchDriver pDriver = pqi;

    HRESULT hr;
    _variant_t pOutVal;

    CoInitialize(NULL);
    hr = pWord.CoCreateInstance(L"Word.Application", NULL, CLSCTX_LOCAL_SERVER);
    if(FAILED(hr)){ /*... handle hr error*/ }

    // Make Word visible.
    hr = pWord.PutPropertyByName(_bstr_t("Visible"), &_variant_t(1));
    if(FAILED(hr)){ /*... handle hr error*/ }

    // Get the Documents collection and store it in new CComPtr
    hr = pWord.GetPropertyByName(_bstr_t("Documents"), &pOutVal);
    if(FAILED(hr)){ /*... handle hr error*/ }

    CComPtr<IDispatch> pDocuments = pOutVal.pdispVal;

    // Use Documents to open a document
    hr = pDocuments.Invoke1 (_bstr_t("Open"),
    &_variant_t("c:\\users\\public\\documents\\sometext.txt"),&pOutVal);
    if(FAILED(hr)){ /*... handle hr error*/ }

    CoUninitialize();
}

```

Consulte também

[Ponteiros inteligentes \(C++ moderno\)](#)

Pimpl para encapsulamento do tempo de compilação (C++ moderno)

02/12/2019 • 2 minutes to read • [Edit Online](#)

O *idioma pimpl* é uma técnica C++ moderna para ocultar a implementação, para minimizar o acoplamento e para separar interfaces. Pimpl é curto para "ponteiro para implementação". Talvez você já esteja familiarizado com o conceito, mas conheça-o por outros nomes, como Cheshire Cat ou idioma do firewall do compilador.

Por que usar o pimpl?

Veja como o idioma pimpl pode melhorar o ciclo de vida do desenvolvimento de software:

- Minimização de dependências de compilação.
- Separação de interface e implementação.
- Portabilidade.

Cabeçalho Pimpl

```
// my_class.h
class my_class {
    // ... all public and protected stuff goes here ...
private:
    class impl; unique_ptr<impl> pimpl; // opaque type here
};
```

O idioma pimpl evita a recompilação em cascata e layouts de objeto frágil. Ele é adequado para tipos populares (transitivamente).

Implementação de Pimpl

Defina a classe `impl` no arquivo. cpp.

```
// my_class.cpp
class my_class::impl { // defined privately here
    // ... all private data and functions: all of these
    //      can now change without recompiling callers ...
};

my_class::my_class(): pimpl( new impl )
{
    // ... set impl values ...
}
```

Práticas recomendadas

Considere se é para adicionar suporte para especialização de permuta sem lançamento.

Consulte também

[Bem-vindo de volta para C++](#)

Referência da linguagem C++

Biblioteca padrão C++

Tratamento de exceção em MSVC

15/04/2020 • 3 minutes to read • [Edit Online](#)

Uma exceção é uma condição de erro, possivelmente fora do controle do programa, que impede que o programa continue sua execução normal. Certas operações, incluindo criação de objetos, entrada/saída de arquivos e chamadas de função feitas a partir de outros módulos, são todas fontes potenciais de exceções, mesmo quando seu programa está sendo executado corretamente. O código robusto antecipa e trata exceções. Para detectar erros lógicos, use afirmações em vez de exceções (consulte [Usando afirmações](#)).

Tipos de exceções

O compilador Microsoft C++ (MSVC) suporta três tipos de tratamento de exceção:

- [C++ manipulação de exceções](#)

Para a maioria dos programas C++, você deve usar o manuseio de exceção C++. É um tipo seguro, e garante que os destruidores de objetos sejam invocados durante o desenrolar da pilha.

- [Tratamento estruturado de exceções](#)

O Windows fornece seu próprio mecanismo de exceção, chamado de tratamento estruturado de exceções (SEH). Não é recomendado para programação C++ ou MFC. Use o SEH apenas em programas que não sejam em MFC ou C.

- [Exceções do MFC:](#)

Desde a versão 3.0, o MFC tem usado exceções C++. Ele ainda suporta suas macros de manuseio de exceção mais antigas, que são semelhantes às exceções C++ na forma. Para obter conselhos sobre a mistura de macros MFC e exceções C++, consulte [Exceções: Usando Macros MFC e Exceções C++](#).

Use uma opção de compilador [/EH](#) para especificar o modelo de manipulação de exceção a ser usado em um projeto C++. Standard C++ exception handling ([/EHsc](#)) é o padrão em novos projetos C++ no Visual Studio.

Não recomendamos que você misture os mecanismos de manipulação de exceção. Por exemplo, não use exceções C++ com tratamento estruturado de exceções. O uso do manuseio de exceção C++ torna exclusivamente o seu código mais portátil, e permite lidar com exceções de qualquer tipo. Para obter mais informações sobre as desvantagens do tratamento estruturado de exceções, consulte [Tratamento estruturado de exceções](#).

Nesta seção

- [Práticas recomendadas modernas do C++ para exceções e manipulação de erros](#)
- [Como projetar tendo em vista a segurança da exceção](#)
- [Como realizar a interface entre códigos excepcional e não excepcional](#)
- [As instruções try, catch e throw](#)
- [Como os blocos catch são avaliados](#)
- [Exceções e Desenrolar stack](#)
- [Especificações de exceção](#)

- `noexcept`
- Exceções C++ não tratadas
- Combinação de exceções C (estruturadas) e C++
- SEH (tratamento de exceções estruturado) (C/C++)

Confira também

[Referência de linguagem C++](#)

[Tratamento de exceções x64](#)

[Tratamento de exceções \(C++/CLI e C++/CX\)](#)

Práticas recomendadas do C++ moderno para exceções e tratamento de erros

02/09/2020 • 13 minutes to read • [Edit Online](#)

No C++ moderno, na maioria dos cenários, a maneira preferida de relatar e lidar com erros lógicos e erros de tempo de execução é usar exceções. Ele é especialmente verdadeiro quando a pilha pode conter várias chamadas de função entre a função que detecta o erro e a função que tem o contexto para manipular o erro. As exceções fornecem uma maneira formal e bem definida para o código que detecta erros para passar as informações para a pilha de chamadas.

Usar exceções para código excepcional

Erros de programa geralmente são divididos em duas categorias: erros lógicos causados por erros de programação, por exemplo, um erro de "índice fora do intervalo". E os erros de tempo de execução que estão além do controle do programador, por exemplo, um erro "serviço de rede indisponível". Na programação em estilo C e no COM, o relatório de erros é gerenciado retornando um valor que representa um código de erro ou um código de status para uma função específica, ou definindo uma variável global que o chamador pode opcionalmente recuperar depois de cada chamada de função para ver se erros foram relatados. Por exemplo, a programação COM usa o valor de retorno HRESULT para comunicar erros ao chamador. E a API do Win32 tem a `GetLastError` função para recuperar o último erro relatado pela pilha de chamadas. Em ambos os casos, cabe ao chamador reconhecer o código e respondê-lo adequadamente. Se o chamador não tratar explicitamente o código de erro, o programa poderá falhar sem aviso. Ou, ele pode continuar a ser executado usando dados incorretos e produzir resultados incorretos.

As exceções são preferenciais no C++ moderno pelos seguintes motivos:

- Uma exceção força o código de chamada a reconhecer uma condição de erro e tratá-la. As exceções não tratadas param a execução do programa.
- Uma exceção salta para o ponto na pilha de chamadas que pode manipular o erro. As funções intermediárias podem permitir que a exceção seja propagada. Eles não precisam coordenar com outras camadas.
- O mecanismo de desenrolamento de pilha de exceção destrói todos os objetos no escopo após a geração de uma exceção, de acordo com as regras bem definidas.
- Uma exceção permite uma separação clara entre o código que detecta o erro e o código que manipula o erro.

O exemplo simplificado a seguir mostra a sintaxe necessária para lançar e capturar exceções em C++.

```

#include <stdexcept>
#include <limits>
#include <iostream>

using namespace std;

void MyFunc(int c)
{
    if (c > numeric_limits<char> ::max())
        throw invalid_argument("MyFunc argument too large.");
    //...
}

int main()
{
    try
    {
        MyFunc(256); //cause an exception to throw
    }

    catch (invalid_argument& e)
    {
        cerr << e.what() << endl;
        return -1;
    }
    //...
    return 0;
}

```

As exceções em C++ parecem aquelas em linguagens como C# e Java. No `try` bloco, se uma exceção for gerada, ela será *detectada* pelo primeiro bloco associado `catch` cujo tipo corresponde ao da exceção. Em outras palavras, a execução salta da `throw` instrução para a `catch` instrução. Se nenhum bloco `catch` utilizável for encontrado, `std::terminate` é invocado e o programa é encerrado. Em C++, qualquer tipo pode ser lançado; no entanto, é recomendável que você gere um tipo que derive direta ou indiretamente do `std::exception`. No exemplo anterior, o tipo de exceção, `invalid_argument`, é definido na biblioteca padrão no arquivo de `<stdexcept>` cabeçalho. O C++ não fornece ou exige um `finally` bloco para garantir que todos os recursos sejam liberados se uma exceção for lançada. O idioma de RAII (aquisição de recursos é inicialização), que usa ponteiros inteligentes, fornece a funcionalidade necessária para a limpeza de recursos. Para obter mais informações, consulte [como: design para segurança de exceção](#). Para obter informações sobre o mecanismo de desenrolamento de pilha C++, consulte [exceções e desenrolamento de pilha](#).

Diretrizes básicas

O tratamento de erros robusto é desafiador em qualquer linguagem de programação. Embora as exceções forneçam vários recursos que dão suporte a um bom tratamento de erros, elas não podem fazer todo o trabalho para você. Para obter os benefícios do mecanismo de exceção, mantenha as exceções em mente ao projetar seu código.

- Use declarações para verificar se há erros que nunca devem ocorrer. Use exceções para verificar se há erros que podem ocorrer, por exemplo, erros na validação de entrada em parâmetros de funções públicas. Para obter mais informações, consulte a seção [exceções versus asserções](#).
- Use exceções quando o código que manipula o erro é separado do código que detecta o erro por uma ou mais chamadas de função intermediárias. Considere se os códigos de erro devem ser usados em loops críticos de desempenho, quando o código que manipula o erro está rigidamente acoplado ao código que o detecta.
- Para cada função que pode lançar ou propagar uma exceção, forneça uma das três garantias de exceção: a garantia forte, a garantia básica ou a garantia `nothrow` (`noexcept`). Para obter mais informações, consulte

como: design para segurança de exceção.

- Lançar exceções por valor, capturá-las por referência. Não pegue o que não é possível manipular.
- Não use especificações de exceção, que são preferidas no C++ 11. Para obter mais informações, consulte as especificações e `noexcept` a seção de exceção.
- Use tipos de exceção de biblioteca padrão quando eles se aplicarem. Derive tipos de exceção personalizados da hierarquia de `exception classes`.
- Não permita que exceções escapem de destruidores ou de funções de desalocação de memória.

Exceções e desempenho

O mecanismo de exceção tem um custo de desempenho mínimo se nenhuma exceção for gerada. Se uma exceção for lançada, o custo da passagem de pilha e o desenrolamento serão aproximadamente comparáveis ao custo de uma chamada de função. Estruturas de dados adicionais são necessárias para rastrear a pilha de chamadas depois que um `try` bloco é inserido e instruções adicionais são necessárias para desenrolar a pilha se uma exceção for lançada. No entanto, na maioria dos cenários, o custo no desempenho e no volume de memória não é significativo. O efeito adverso de exceções no desempenho provavelmente será significativo apenas em sistemas com restrição de memória. Ou, em loops de desempenho crítico, em que é provável que um erro ocorra regularmente e haja acoplamento forte entre o código para tratá-lo e o código que o relata. Em qualquer caso, é impossível saber o custo real das exceções sem a criação de perfil e a medição. Mesmo nesses casos raros, quando o custo é significativo, você pode pesar isso com a maior exatidão, facilidade de manutenção e outras vantagens que são fornecidas por uma política de exceção bem projetada.

Exceções versus asserções

Exceções e declarações são dois mecanismos distintos para detectar erros em tempo de execução em um programa. Use `assert` instruções para testar condições durante o desenvolvimento que nunca devem ser verdadeiras se todo o código estiver correto. Não há nenhum ponto em lidar com esse erro usando uma exceção, porque o erro indica que algo no código deve ser corrigido. Ele não representa uma condição que o programa precisa recuperar em tempo de execução. Um `assert` interrompe a execução na instrução para que você possa inspecionar o estado do programa no depurador. Uma exceção continua a execução do primeiro manipulador `catch` apropriado. Use exceções para verificar condições de erro que podem ocorrer em tempo de execução, mesmo que seu código esteja correto, por exemplo, "arquivo não encontrado" ou "memória insuficiente". As exceções podem lidar com essas condições, mesmo que a recuperação apenas gere uma mensagem para um log e encerre o programa. Sempre verifique os argumentos para funções públicas usando exceções. Mesmo que sua função seja sem erros, talvez você não tenha controle total sobre os argumentos que um usuário pode passar para ele.

Exceções de C++ versus exceções de SEH do Windows

Os programas C e C++ podem usar o mecanismo de manipulação de exceção estruturada (SEH) no sistema operacional Windows. Os conceitos em SEH se assemelham àqueles em exceções C++, exceto que o SEH usa as `__try` `__except` construções, e `__finally` em vez de `try` e `catch`. No compilador do Microsoft C++ (MSVC), as exceções do C++ são implementadas para SEH. No entanto, ao escrever código C++, use a sintaxe de exceção C++.

Para obter mais informações sobre SEH, consulte [manipulação de exceção estruturada \(C/C++\)](#).

Especificações de exceção e `noexcept`

As especificações de exceção foram introduzidas em C++ como uma maneira de especificar as exceções que

uma função pode gerar. No entanto, as especificações de exceção provaram problemas na prática e foram preteridas no padrão de rascunho do C++ 11. Recomendamos que você não use as `throw` especificações de exceção `throw()`, exceto para, o que indica que a função não permite a saída de nenhuma exceção. Se você precisar usar especificações de exceção da forma preterida `throw(type-name)`, o suporte a MSVC será limitado. Para obter mais informações, consulte [especificações de exceção \(throw\)](#). O `noexcept` especificador é introduzido no C++ 11 como a alternativa preferida ao `throw()`.

Confira também

[Como: interface entre códigos excepcionais e não excepcionais](#)

[Referência da linguagem C++](#)

[Biblioteca padrão do C++](#)

Como criar para segurança de exceção

02/09/2020 • 12 minutes to read • [Edit Online](#)

Uma das vantagens do mecanismo de exceção é que a execução, junto com os dados sobre a exceção, salta diretamente da instrução que gera a exceção para a primeira instrução catch que a manipula. O manipulador pode ser qualquer número de níveis na pilha de chamadas. Funções que são chamadas entre a instrução try e a instrução Throw não são necessárias para saber nada sobre a exceção que é lançada. No entanto, eles precisam ser projetados para que possam sair do escopo "inesperadamente" em qualquer ponto em que uma exceção possa se propagar abaixo e fazer isso sem deixar por trás de objetos criados parcialmente, memória vazada ou estruturas de dados que estejam em Estados inutilizáveis.

Técnicas básicas

Uma política de tratamento de exceção robusta requer uma consideração cuidadosa e deve fazer parte do processo de design. Em geral, a maioria das exceções são detectadas e geradas nas camadas inferiores de um módulo de software, mas normalmente essas camadas não têm contexto suficiente para lidar com o erro ou expor uma mensagem aos usuários finais. Nas camadas intermediárias, as funções podem capturar e relançar uma exceção quando precisam inspecionar o objeto de exceção ou têm informações úteis adicionais para fornecer a camada superior que, por fim, captura a exceção. Uma função deve capturar e "assimilar" uma exceção somente se for capaz de se recuperar por completo dela. Em muitos casos, o comportamento correto nas camadas intermediárias é permitir que uma exceção propague a pilha de chamadas. Mesmo na camada mais alta, pode ser apropriado permitir que uma exceção sem tratamento encerre um programa se a exceção deixar o programa em um estado no qual sua exatidão não possa ser garantida.

Não importa como uma função manipula uma exceção, para ajudar a garantir que ela seja "segura de exceção", ela deve ser projetada de acordo com as regras básicas a seguir.

Manter classes de recursos simples

Quando você encapsula o gerenciamento manual de recursos em classes, use uma classe que não faz nada, exceto gerenciar um único recurso. Ao manter a classe simples, você reduz o risco de introduzir vazamentos de recursos. Use [ponteiros inteligentes](#) quando possível, conforme mostrado no exemplo a seguir. Este exemplo é intencionalmente artificial e simplista para destacar as diferenças quando `shared_ptr` é usado.

```

// old-style new/delete version
class NDResourceClass {
private:
    int*    m_p;
    float*  m_q;
public:
    NDResourceClass() : m_p(0), m_q(0) {
        m_p = new int;
        m_q = new float;
    }

    ~NDResourceClass() {
        delete m_p;
        delete m_q;
    }
    // Potential leak! When a constructor emits an exception,
    // the destructor will not be invoked.
};

// shared_ptr version
#include <memory>

using namespace std;

class SPResourceClass {
private:
    shared_ptr<int> m_p;
    shared_ptr<float> m_q;
public:
    SPResourceClass() : m_p(new int), m_q(new float) { }
    // Implicitly defined dtor is OK for these members,
    // shared_ptr will clean up and avoid leaks regardless.
};

// A more powerful case for shared_ptr

class Shape {
    // ...
};

class Circle : public Shape {
    // ...
};

class Triangle : public Shape {
    // ...
};

class SPSHapeResourceClass {
private:
    shared_ptr<Shape> m_p;
    shared_ptr<Shape> m_q;
public:
    SPSHapeResourceClass() : m_p(new Circle), m_q(new Triangle) { }
};

```

Usar o idioma RAI^I para gerenciar recursos

Para ser seguro de exceção, uma função deve garantir que os objetos alocados usando `malloc` ou `new` sejam destruídos, e todos os recursos, como os identificadores de arquivo, sejam fechados ou liberados, mesmo que uma exceção seja lançada. O idioma de RAI^I (*quisição de recursos é inicialização*) une o gerenciamento desses recursos ao ciclo de vida de variáveis automáticas. Quando uma função sai do escopo, retornando normalmente ou devido a uma exceção, os destruidores para todas as variáveis automáticas totalmente construídas são invocados. Um objeto wrapper RAI^I, como um ponteiro inteligente, chama a função Delete ou Close apropriada em seu destruidor. No código de exceção segura, é extremamente importante passar a propriedade de cada

recurso imediatamente para algum tipo de objeto RAI. Observe que as `vector`, `string`, `classes`, `make_shared`, `fstream` e semelhantes manipulam a aquisição do recurso para você. No entanto, `unique_ptr` e as `shared_ptr` construções tradicionais são especiais porque a aquisição de recursos é executada pelo usuário em vez do objeto; portanto, elas contam como *liberação de recursos é a destruição*, mas é questionável como RAI.

As três garantias de exceção

Normalmente, a segurança de exceção é discutida em termos das três garantias que uma função pode fornecer: a *garantia sem falha*, a *garantia forte* e a *garantia básica*.

Garantia sem falha

A garantia sem falha (ou, "no-throw") é a garantia mais forte que uma função pode fornecer. Ele informa que a função não lançará uma exceção ou permitirá que uma seja propagada. No entanto, você não pode fornecer de forma confiável tal garantia, a menos que (a) você saiba que todas as funções que essa função chama também não têm falha, ou (b) você sabe que todas as exceções que são geradas são detectadas antes que elas cheguem a essa função, ou (c) você sabe como capturar e lidar corretamente com todas as exceções que possam alcançar essa.

Tanto a garantia forte quanto a garantia básica dependem da suposição de que os destruidores não são-falham. Todos os contêineres e tipos na biblioteca padrão garantem que seus destruidores não lançam. Também há um requisito de inverso: a biblioteca padrão requer que os tipos definidos pelo usuário que são fornecidos a ele — por exemplo, como argumentos de modelo — devem ter destruidores sem lançamento.

Alta garantia

A alta garantia indica que, se uma função sair do escopo devido a uma exceção, ela não vazará memória e o estado do programa não será modificado. Uma função que fornece uma forte garantia é, essencialmente, uma transação com semântica de confirmação ou reversão: ela é completamente sucedido ou não tem efeito.

Garantia básica

A garantia básica é a mais fraca dos três. No entanto, pode ser a melhor opção quando uma alta garantia é muito cara no consumo de memória ou no desempenho. A garantia básica indica que, se ocorrer uma exceção, nenhuma memória será vazada e o objeto ainda estará em um estado utilizável, mesmo que os dados possam ter sido modificados.

Classes de exceção segura

Uma classe pode ajudar a garantir sua própria segurança de exceção, mesmo quando ela é consumida por funções inseguras, impedindo que ela seja parcialmente construída ou parcialmente destruída. Se um construtor de classe sair antes da conclusão, o objeto nunca será criado e seu destruidor nunca será chamado. Embora as variáveis automáticas que são inicializadas antes da exceção tenham seus destruidores invocados, a memória alocada dinamicamente ou os recursos que não são gerenciados por um ponteiro inteligente ou variável automática semelhante serão vazados.

Os tipos internos são todos sem falha e os tipos de biblioteca padrão oferecem suporte à garantia básica no mínimo. Siga estas diretrizes para qualquer tipo definido pelo usuário que deve ser seguro para exceções:

- Use apontadores inteligentes ou outros wrappers de tipo RAI para gerenciar todos os recursos. Evite a funcionalidade de gerenciamento de recursos em seu destruidor de classe, pois o destruidor não será invocado se o Construtor lançar uma exceção. No entanto, se a classe for um Gerenciador de recursos dedicado que controla apenas um recurso, é aceitável usar o destruidor para gerenciar recursos.
- Entenda que uma exceção gerada em um construtor de classe base não pode ser assimilada em um construtor de classe derivada. Se você quiser converter e relançar a exceção de classe base em um Construtor derivado, use um bloco try de função.

- Considere a possibilidade de armazenar todo o estado da classe em um membro de dados que esteja encapsulado em um ponteiro inteligente, especialmente se uma classe tiver um conceito de "inicialização com permissão para falhar". Embora o C++ permita membros de dados não inicializados, ele não oferece suporte a instâncias de classe não inicializadas ou parcialmente iniciadas. Um construtor deve ser bem-sucedida ou falhar; nenhum objeto será criado se o construtor não for executado até a conclusão.
- Não permita que nenhuma exceção escape de um destruidor. Um axioma básico do C++ é que os destruidores nunca devem permitir que uma exceção propague a pilha de chamadas. Se um destruidor precisar executar uma operação de lançamento de exceção potencialmente, ele deverá fazer isso em um bloco try catch e assimilar a exceção. A biblioteca padrão fornece essa garantia em todos os destruidores que ele define.

Confira também

[Práticas recomendadas do C++ moderno para exceções e tratamento de erros](#)

[Como: interface entre códigos excepcionais e não excepcionais](#)

Como: interface entre códigos excepcionais e não excepcionais

02/09/2020 • 10 minutes to read • [Edit Online](#)

Este artigo descreve como implementar o tratamento de exceção consistente em um módulo C++ e também como converter essas exceções de e para códigos de erro nos limites de exceção.

Às vezes, um módulo C++ tem de se fazer interface com código que não usa exceções (código não excepcional). Essa interface é conhecida como limite de exceção. Por exemplo, talvez você queira chamar a função do Win32 `CreateFile` em seu programa C++. `CreateFile` não gera exceções; em vez disso, ele define códigos de erro que podem ser recuperados pela `GetLastError` função. Se o seu programa em C++ não for trivial, você provavelmente prefere ter uma política de tratamento de erros baseada em exceção consistente. E você provavelmente não desejará abandonar exceções apenas porque você se baseia em uma interface com código não-excepcional e nenhuma delas deseja misturar políticas de erro baseadas em exceção e não baseadas em exceção em seu módulo C++.

Chamando funções não excepcionais do C++

Quando você chama uma função não excepcional do C++, a ideia é encapsular essa função em uma função C++ que detecta erros e, em seguida, gera uma exceção. Quando você projeta essa função de invólucro, primeiro decida qual tipo de garantia de exceção fornecer: no-throw, strong ou Basic. Em segundo lugar, projete a função para que todos os recursos, por exemplo, identificadores de arquivo, sejam liberados corretamente se uma exceção for lançada. Normalmente, isso significa que você usa apontadores inteligentes ou gerenciadores de recursos semelhantes para ter os recursos. Para obter mais informações sobre considerações de design, consulte [como: design para segurança de exceção](#).

Exemplo

O exemplo a seguir mostra as funções do C++ que usam o Win32 `CreateFile` e as `ReadFile` funções internamente para abrir e ler dois arquivos. A `File` classe é um wrapper de RAII (aquisição de recursos é inicialização) para os identificadores de arquivo. Seu Construtor detecta uma condição "arquivo não encontrado" e gera uma exceção para propagar o erro na pilha de chamadas do módulo C++ (neste exemplo, a `main()` função). Se uma exceção for lançada depois `File` que um objeto for totalmente construído, o destruidor chamará automaticamente `CloseHandle` para liberar o identificador de arquivo. (Se preferir, você pode usar a classe Active Template Library (ATL) `CHandle` para essa mesma finalidade ou uma `unique_ptr` com uma excluidor personalizada.) As funções que chamam APIs Win32 e CRT detectam erros e, em seguida, geram exceções em C++ usando a função definida localmente `ThrowLastErrorIf`, que, por sua vez, usa a `Win32Exception` classe, derivada da `runtime_error` classe. Todas as funções neste exemplo fornecem uma forte garantia de exceção; se uma exceção for lançada em qualquer ponto dessas funções, nenhum recurso será vazado e nenhum estado do programa é modificado.

```
// compile with: /EHsc
#include <Windows.h>
#include <stdlib.h>
#include <vector>
#include <iostream>
#include <string>
#include <limits>
#include <stdexcept>

using namespace std;
```

```

string FormatErrorMessage(DWORD error, const string& msg)
{
    static const int BUFFERLENGTH = 1024;
    vector<char> buf(BUFFERLENGTH);
    FormatMessageA(FORMAT_MESSAGE_FROM_SYSTEM, 0, error, 0, buf.data(),
        BUFFERLENGTH - 1, 0);
    return string(buf.data()) + " (" + msg + ")";
}

class Win32Exception : public runtime_error
{
private:
    DWORD m_error;
public:
    Win32Exception(DWORD error, const string& msg)
        : runtime_error(FormatErrorMessage(error, msg)), m_error(error) { }

    DWORD GetErrorCode() const { return m_error; }
};

void ThrowLastErrorIf(bool expression, const string& msg)
{
    if (expression) {
        throw Win32Exception(GetLastError(), msg);
    }
}

class File
{
private:
    HANDLE m_handle;

    // Declared but not defined, to avoid double closing.
    File& operator=(const File&);
    File(const File&);

public:
    explicit File(const string& filename)
    {
        m_handle = CreateFileA(filename.c_str(), GENERIC_READ, FILE_SHARE_READ,
            nullptr, OPEN_EXISTING, FILE_ATTRIBUTE_READONLY, nullptr);
        ThrowLastErrorIf(m_handle == INVALID_HANDLE_VALUE,
            "CreateFile call failed on file named " + filename);
    }

    ~File() { CloseHandle(m_handle); }

    HANDLE GetHandle() { return m_handle; }
};

size_t GetFileSizeSafe(const string& filename)
{
    File fobj(filename);
    LARGE_INTEGER filesize;

    BOOL result = GetFileSizeEx(fobj.GetHandle(), &filesize);
    ThrowLastErrorIf(result == FALSE, "GetFileSizeEx failed: " + filename);

    if (filesize.QuadPart < (numeric_limits<size_t>::max)()) {
        return filesize.QuadPart;
    } else {
        throw;
    }
}

vector<char> ReadFileVector(const string& filename)
{
    File fobj(filename);
    size_t filesize = GetFileSizeSafe(filename);
    DWORD bytesRead = 0;

```

```

DWORD bytesRead = 0,
vector<char> readbuffer(filesize);

BOOL result = ReadFile(fobj.GetHandle(), readbuffer.data(), readbuffer.size(),
&bytesRead, nullptr);
ThrowLastErrorIf(result == FALSE, "ReadFile failed: " + filename);

cout << filename << " file size: " << filesize << ", bytesRead: "
<< bytesRead << endl;

return readbuffer;
}

bool IsFileDiff(const string& filename1, const string& filename2)
{
    return ReadFileVector(filename1) != ReadFileVector(filename2);
}

#include <iomanip>

int main ( int argc, char* argv[] )
{
    string filename1("file1.txt");
    string filename2("file2.txt");

    try
    {
        if(argc > 2) {
            filename1 = argv[1];
            filename2 = argv[2];
        }

        cout << "Using file names " << filename1 << " and " << filename2 << endl;

        if (IsFileDiff(filename1, filename2)) {
            cout << "++ Files are different." << endl;
        } else {
            cout << "== Files match." << endl;
        }
    }
    catch(const Win32Exception& e)
    {
        ios state(nullptr);
        state.copyfmt(cout);
        cout << e.what() << endl;
        cout << "Error code: 0x" << hex << uppercase << setw(8) << setfill('0')
            << e.GetErrorCode() << endl;
        cout.copyfmt(state); // restore previous formatting
    }
}
}

```

Chamando código excepcional de código não-excepcional

As funções do C++ que são declaradas como "extern C" podem ser chamadas por programas do C. Os servidores COM C++ podem ser consumidos pelo código escrito em qualquer um de vários idiomas diferentes. Quando você implementa funções públicas com reconhecimento de exceção em C++ para serem chamadas por código não-excepcional, a função C++ não deve permitir que nenhuma exceção seja propagada de volta para o chamador. Portanto, a função C++ deve capturar especificamente todas as exceções que ele saiba como manipular e, se apropriado, converter a exceção em um código de erro que o chamador entenda. Se nem todas as possíveis exceções forem conhecidas, a função C++ deverá ter um `catch(...)` bloco como o último manipulador. Nesse caso, é melhor relatar um erro fatal para o chamador, pois seu programa pode estar em um estado desconhecido.

O exemplo a seguir mostra uma função que supõe que qualquer exceção que possa ser gerada é uma

Win32exception ou um tipo de exceção derivado de `std::exception`. A função captura qualquer exceção desses tipos e propaga as informações de erro como um código de erro do Win32 para o chamador.

```
BOOL DiffFiles2(const string& file1, const string& file2)
{
    try
    {
        File f1(file1);
        File f2(file2);
        if (IsTextFileDiff(f1, f2))
        {
            SetLastError(MY_APPLICATION_ERROR_FILE_MISMATCH);
            return FALSE;
        }
        return TRUE;
    }
    catch(Win32Exception& e)
    {
        SetLastError(e.GetErrorCode());
    }

    catch(std::exception& e)
    {
        SetLastError(MY_APPLICATION_GENERAL_ERROR);
    }
    return FALSE;
}
```

Quando você converte de exceções para códigos de erro, um possível problema é que os códigos de erro geralmente não contêm a riqueza de informações que uma exceção pode armazenar. Para resolver isso, você pode fornecer um `catch` bloco para cada tipo de exceção específico que pode ser lançado e executar o registro em log para registrar os detalhes da exceção antes que ela seja convertida em um código de erro. Essa abordagem pode criar muita repetição de código se várias funções usarem o mesmo conjunto de `catch` blocos. Uma boa maneira de evitar a repetição de código é refatorar esses blocos em uma função de utilitário particular que implementa `try` os `catch` blocos e aceita um objeto de função que é invocado no `try` bloco. Em cada função pública, passe o código para a função do utilitário como uma expressão lambda.

```
template<typename Func>
bool Win32ExceptionBoundary(Func&& f)
{
    try
    {
        return f();
    }
    catch(Win32Exception& e)
    {
        SetLastError(e.GetErrorCode());
    }
    catch(const std::exception& e)
    {
        SetLastError(MY_APPLICATION_GENERAL_ERROR);
    }
    return false;
}
```

O exemplo a seguir mostra como gravar a expressão lambda que define o functor. Quando um functor é definido "inline" usando uma expressão lambda, geralmente é mais fácil de ler do que seria se ele fosse gravado como um objeto de função nomeado.

```
bool DiffFiles3(const string& file1, const string& file2)
{
    return Win32ExceptionBoundary([&]() -> bool
    {
        File f1(file1);
        File f2(file2);
        if (IsTextFileDiff(f1, f2))
        {
            SetLastError(MY_APPLICATION_ERROR_FILE_MISMATCH);
            return false;
        }
        return true;
    });
}
```

Para obter mais informações sobre expressões lambda, consulte [expressões lambda](#).

Confira também

[Práticas recomendadas do C++ moderno para exceções e tratamento de erros](#)

[Como criar para segurança de exceção](#)

Instruções try, throw e catch (C++)

02/09/2020 • 5 minutes to read • [Edit Online](#)

Para implementar a manipulação de exceção em C++, você usa `try` `throw` expressões, e `catch`.

Primeiro, use um `try` bloco para incluir uma ou mais instruções que podem gerar uma exceção.

Uma `throw` expressão sinaliza que uma condição excepcional – muitas vezes, um erro — ocorreu em um `try` bloco. Você pode usar um objeto de qualquer tipo como o operando de uma `throw` expressão. Normalmente, esse objeto é usado para passar informações sobre o erro. Na maioria dos casos, recomendamos que você use a classe `std::Exception` ou uma das classes derivadas que são definidas na biblioteca padrão. Se uma delas não for apropriada, recomendamos que você derive sua própria classe de exceção de `std::exception`.

Para lidar com exceções que podem ser geradas, implemente um ou mais `catch` blocos imediatamente após um `try` bloco. Cada `catch` bloco especifica o tipo de exceção que pode manipular.

Este exemplo mostra um `try` bloco e seus manipuladores. Suponhamos que `GetNetworkResource()` adquira dados por uma conexão de rede e que os dois tipos de exceções sejam classes definidas pelo usuário que derivam de `std::exception`. Observe que as exceções são capturadas por `const` referência na `catch` instrução. Recomendamos que você lance exceções por valor e capture-as pela referência `const`.

Exemplo

```
MyData md;
try {
    // Code that could throw an exception
    md = GetNetworkResource();
}
catch (const networkIOException& e) {
    // Code that executes when an exception of type
    // networkIOException is thrown in the try block
    // ...
    // Log error message in the exception object
    cerr << e.what();
}
catch (const myDataFormatException& e) {
    // Code that handles another exception type
    // ...
    cerr << e.what();
}

// The following syntax shows a throw expression
MyData GetNetworkResource()
{
    // ...
    if (IOSuccess == false)
        throw networkIOException("Unable to connect");
    // ...
    if (readError)
        throw myDataFormatException("Format error");
    // ...
}
```

Comentários

O código após a `try` cláusula é a seção de código protegida. A `throw` expressão *gera*— ou seja, gera — uma exceção. O bloco de código após a `catch` cláusula é o manipulador de exceção. Esse é o manipulador que *captura* a exceção que é gerada se os tipos nas `throw` `catch` expressões e são compatíveis. Para obter uma lista de regras que regem a correspondência de tipos em `catch` blocos, consulte [como os blocos de captura são avaliados](#). Se a `catch` instrução especificar reticências (...) em vez de um tipo, o `catch` bloco tratará de cada tipo de exceção. Quando você compila com a opção `/EHA`, elas podem incluir exceções estruturadas C e exceções assíncronas geradas pelo sistema ou pelo aplicativo, como proteção de memória, divisão por zero e violações de ponto flutuante. Como `catch` blocos são processados em ordem de programa para localizar um tipo correspondente, um manipulador de reticências deve ser o último manipulador para o `try` bloco associado. Use `catch(...)` com cuidado; não permita que um programa continue a menos que o bloco `catch` saiba tratar a exceção específica que será capturada. Normalmente, um bloco `catch(...)` é usado para registrar erros e executar a limpeza especial antes de execução do programa ser interrompida.

Uma `throw` expressão que não tem nenhum operando lança novamente a exceção que está sendo manipulada no momento. Recomendamos essa forma ao lançar novamente a exceção, pois isso preserva as informações de tipo polimórfico da exceção original. Essa expressão deve ser usada apenas em um `catch` manipulador ou em uma função que é chamada a partir de um `catch` manipulador. O objeto de exceção lançado novamente é o objeto da exceção original, não uma cópia.

```
try {
    throw CSomeOtherException();
}
catch(...) {
    // Catch all exceptions - dangerous!!!
    // Respond (perhaps only partially) to the exception, then
    // re-throw to pass the exception to some other handler
    // ...
    throw;
}
```

Confira também

[Práticas recomendadas do C++ moderno para exceções e tratamento de erros](#)

[Palavras-chave](#)

[Exceções de C++ não tratadas](#)

[_uncaught_exception](#)

Como blocos catch são avaliados (C++)

02/09/2020 • 4 minutes to read • [Edit Online](#)

O C++ permite lançar exceções de qualquer tipo, embora seja recomendável lançar os tipos derivados de `std::exception`. Uma exceção de C++ pode ser detectada por um `catch` manipulador que especifica o mesmo tipo da exceção gerada ou por um manipulador que pode capturar qualquer tipo de exceção.

Se o tipo de exceção lançada for uma classe, que também tenha uma classe base (ou classes), ela pode ser capturada pelos manipuladores que aceitam classes base do tipo da exceção, bem como por referências às bases do tipo da exceção. Observe que, quando uma exceção é capturada por uma referência, ela é associada ao objeto de exceção lançado real; caso contrário, é uma cópia (bem semelhante a um argumento para uma função).

Quando uma exceção é lançada, ela pode ser detectada pelos seguintes tipos de `catch` manipuladores:

- Um manipulador que pode aceitar qualquer tipo (usando a sintaxe de reticências).
- Um manipulador que aceita o mesmo tipo que o objeto de exceção; Porque é uma cópia, `const` e os `volatile` modificadores são ignorados.
- Um manipulador que aceita uma referência para mesmo tipo do objeto de exceção.
- Um manipulador que aceita uma referência a um `const volatile` formulário ou do mesmo tipo que o objeto de exceção.
- Um manipulador que aceita uma classe base do mesmo tipo que o objeto de exceção; Como é uma cópia, `const` e os `volatile` modificadores são ignorados. O `catch` manipulador para uma classe base não deve preceder o `catch` manipulador para a classe derivada.
- Um manipulador que aceita uma referência a uma classe base do mesmo tipo do objeto de exceção.
- Um manipulador que aceita uma referência a um `const volatile` formulário ou de uma classe base do mesmo tipo que o objeto de exceção.
- Um manipulador que aceita um ponteiro no qual um objeto de ponteiro gerado pode ser convertido pelas regras padrão de conversão de ponteiro.

A ordem na qual os `catch` manipuladores aparecem é significativa, pois os manipuladores para um determinado `try` bloco são examinados em ordem de sua aparência. Por exemplo, é um erro para colocar o manipulador de uma classe base antes do manipulador de uma classe derivada. Depois que um `catch` manipulador correspondente é encontrado, os manipuladores subsequentes não são examinados. Como resultado, um manipulador de reticências `catch` deve ser o último manipulador para seu `try` bloco. Por exemplo:

```
// ...
try
{
    // ...
}
catch( ... )
{
    // Handle exception here.
}
// Error: the next two handlers are never examined.
catch( const char * str )
{
    cout << "Caught exception: " << str << endl;
}
catch( CExcptClass E )
{
    // Handle CExcptClass exception here.
}
```

Neste exemplo, o manipulador de reticências `catch` é o único manipulador que é examinado.

Confira também

[Práticas recomendadas do C++ moderno para exceções e tratamento de erros](#)

Exceções e desenrolamento da pilha em C++

02/09/2020 • 7 minutes to read • [Edit Online](#)

No mecanismo de exceção do C++, o controle move-se da instrução `throw` para a primeira instrução `catch` que pode manipular o tipo lançado. Quando a instrução `Catch` é atingida, todas as variáveis automáticas que estão no escopo entre as instruções `throw` e `catch` são destruídas em um processo conhecido como *desenrolamento de pilha*. No desenrolamento de pilha, a execução ocorre da seguinte maneira:

1. O controle alcança a `try` instrução pela execução sequencial normal. A seção protegida no `try` bloco é executada.
2. Se nenhuma exceção for gerada durante a execução da seção protegida, as `catch` cláusulas que seguem o `try` bloco não serão executadas. A execução continua na instrução após a última `catch` cláusula que segue o `try` bloco associado.
3. Se uma exceção for lançada durante a execução da seção protegida ou em qualquer rotina que a seção protegida chama de forma direta ou indireta, um objeto de exceção é criado a partir do objeto criado pelo `throw` operando. (Isso implica que um construtor de cópia pode estar envolvido.) Neste ponto, o compilador procura uma `catch` cláusula em um contexto de execução mais alto que pode manipular uma exceção do tipo que é lançado ou para um `catch` manipulador que pode manipular qualquer tipo de exceção. Os `catch` manipuladores são examinados em ordem de sua aparência após o `try` bloco. Se nenhum manipulador apropriado for encontrado, o próximo bloco delimitador dinamicamente `try` será examinado. Esse processo continua até que o bloco de circunscrição mais externo `try` seja examinado.
4. Se mesmo assim um manipulador correspondente não for localizado, ou se uma exceção ocorrer durante o processo de desenrolamento antes que o manipulador obtenha o controle, a função de tempo de execução predefinida `terminate` é chamada. Se uma exceção ocorrer depois que a exceção for lançada, mas antes do início do desenrolamento, `terminate` é chamada.
5. Se um `catch` manipulador correspondente for encontrado e ele for detectado por valor, seu parâmetro formal será inicializado copiando o objeto de exceção. Se a captura for feita por referência, o parâmetro é inicializado para consultar o objeto de exceção. Depois que o parâmetro formal for inicializado, o processo de desenrolamento de pilha é iniciado. Isso envolve a destruição de todos os objetos automáticos que foram totalmente construídos, mas que ainda não foram destruídos, entre o início do `try` bloco associado ao `catch` manipulador e o site de lançamento da exceção. A destruição ocorre na ordem inversa da construção. O `catch` manipulador é executado e o programa retoma a execução após o último manipulador — ou seja, na primeira instrução ou construção que não é um `catch` manipulador. O controle só pode inserir um `catch` manipulador por meio de uma exceção gerada, nunca por meio de uma `goto` instrução ou um `case` rótulo em uma `switch` instrução.

Exemplo de desenrolamento de pilha

O exemplo a seguir demonstra como a pilha é desenrolada depois que uma exceção é lançada. A execução do thread ignora a instrução `throw` em `C` e passa à instrução `catch` em `main`, desenrolando cada função ao longo do caminho. Observe a ordem na qual os objetos `Dummy` são criados e destruídos à medida que saem do escopo. Observe também que nenhuma função é concluída, exceto `main`, que contém a instrução `catch`. A função `A` nunca retorna da sua chamada a `B()`, e `B` nunca retorna de sua chamada a `C()`. Se você remover o comentário da definição do ponteiro `Dummy` e a instrução `delete` correspondente, e executar o programa em seguida, observará que o ponteiro nunca será excluído. Isso mostra o que pode acontecer quando as funções não fornecem uma garantia de exceção. Para obter mais informações, consulte Instruções: Design para exceções. Se você fizer

comentários fora da instrução catch, observe o que acontece quando um programa é encerrado devido a uma exceção sem tratamento.

```
#include <string>
#include <iostream>
using namespace std;

class MyException{};
class Dummy
{
public:
    Dummy(string s) : MyName(s) { PrintMsg("Created Dummy:"); }
    Dummy(const Dummy& other) : MyName(other.MyName){ PrintMsg("Copy created Dummy:"); }
    ~Dummy(){ PrintMsg("Destroyed Dummy:"); }
    void PrintMsg(string s) { cout << s << MyName << endl; }
    string MyName;
    int level;
};

void C(Dummy d, int i)
{
    cout << "Entering FunctionC" << endl;
    d.MyName = " C";
    throw MyException();

    cout << "Exiting FunctionC" << endl;
}

void B(Dummy d, int i)
{
    cout << "Entering FunctionB" << endl;
    d.MyName = "B";
    C(d, i + 1);
    cout << "Exiting FunctionB" << endl;
}

void A(Dummy d, int i)
{
    cout << "Entering FunctionA" << endl;
    d.MyName = " A" ;
    // Dummy* pd = new Dummy("new Dummy"); //Not exception safe!!!
    B(d, i + 1);
    // delete pd;
    cout << "Exiting FunctionA" << endl;
}

int main()
{
    cout << "Entering main" << endl;
    try
    {
        Dummy d(" M");
        A(d,1);
    }
    catch (MyException& e)
    {
        cout << "Caught an exception of type: " << typeid(e).name() << endl;
    }

    cout << "Exiting main." << endl;
    char c;
    cin >> c;
}

/* Output:
   Entering main
   Created Dummy: M
   Copy created Dummy: M
   Destroyed Dummy: M
   Exiting main.
   M
```

```
Copy created Dummy: M
Entering FunctionA
Copy created Dummy: A
Entering FunctionB
Copy created Dummy: B
Entering FunctionC
Destroyed Dummy: C
Destroyed Dummy: B
Destroyed Dummy: A
Destroyed Dummy: M
Caught an exception of type: class MyException
Exiting main.
```

```
*/
```

Especificações de exceção (throw, noexcept) (C++)

02/09/2020 • 7 minutes to read • [Edit Online](#)

As especificações de exceção são um recurso de linguagem do C++ que indica a intenção do programador sobre os tipos de exceção que podem ser propagados por uma função. Você pode especificar que uma função pode ou não ser encerrada por uma exceção usando uma *especificação de exceção*. O compilador pode usar essas informações para otimizar as chamadas para a função e para encerrar o programa se uma exceção inesperada escapar da função.

Antes do C++ 17 havia dois tipos de especificação de exceção. A *especificação noexcept* era nova no C++ 11. Especifica se o conjunto de possíveis exceções que podem escapar da função está vazio. A *especificação de exceção dinâmica*, ou `throw(optional_type_list)` especificação, foi preterida no C++ 11 e removida no C++ 17, exceto para `throw()`, que é um alias para `noexcept(true)`. Essa especificação de exceção foi criada para fornecer informações resumidas sobre quais exceções podem ser geradas fora de uma função, mas na prática ela foi considerada problemática. A especificação de exceção dinâmica que provava ser um pouco útil era a especificação incondicional `throw()`. Por exemplo, a declaração da função:

```
void MyFunction(int i) throw();
```

informa o compilador que a função não lança exceções. No entanto, em `/std: modo C++ 14` isso poderia levar a um comportamento indefinido se a função gerar uma exceção. Portanto, é recomendável usar o operador `noexcept` em vez de um acima:

```
void MyFunction(int i) noexcept;
```

A tabela a seguir resume a implementação de especificações de exceção do Microsoft C++:

ESPECIFICAÇÃO DE EXCEÇÃO	SIGNIFICADO
<code>noexcept</code> <code>noexcept(true)</code> <code>throw()</code>	A função não gera uma exceção. Em <code>/std: modo C++ 14</code> (que é o padrão) <code>noexcept</code> e <code>noexcept(true)</code> são equivalentes. Quando uma exceção é gerada de uma função que é declarada <code>noexcept</code> ou <code>noexcept(true)</code> , <code>std::Terminate</code> é invocado. Quando uma exceção é gerada de uma função declarada como <code>throw()</code> no modo <code>/std: C++ 14</code> , o resultado é um comportamento indefinido. Nenhuma função específica é invocada. Essa é uma divergência do padrão C++ 14, que exigia que o compilador invocasse <code>std::inesperado</code> . Visual Studio 2017 versão 15,5 e posterior: em <code>/std: modo C++ 17</code> , <code>noexcept</code> , <code>noexcept(true)</code> e <code>throw()</code> todos são equivalentes. Em <code>/std: modo C++ 17</code> , <code>throw()</code> é um alias para <code>noexcept(true)</code> . Em <code>/std: modo C++ 17</code> , quando uma exceção é gerada de uma função declarada com qualquer uma dessas especificações, <code>std::Terminate</code> é invocado conforme exigido pelo padrão C++ 17.
<code>noexcept(false)</code> <code>throw(...)</code> Sem especificação	A função pode gerar uma exceção de qualquer tipo.

ESPECIFICAÇÃO DE EXCEÇÃO	SIGNIFICADO
<code>throw(type)</code>	(C++ 14 e anterior) A função pode gerar uma exceção do tipo <code>type</code> . O compilador aceita a sintaxe, mas a interpreta como <code>noexcept(false)</code> . Em /std: modo c++ 17 o compilador emite aviso C5040.

Se o tratamento de exceção for usado em um aplicativo, deve haver uma função na pilha de chamadas que manipula exceções lançadas antes de sair do escopo externo de uma função marcada `noexcept`, `noexcept(true)` ou `throw()`. Se qualquer função chamada entre aquela que gera uma exceção e a que manipula a exceção for especificada como `noexcept` `noexcept(true)` (ou `throw()` no modo /std: c++ 17), o programa será encerrado quando a função noexcept propagar a exceção.

O comportamento de exceção de uma função depende dos seguintes fatores:

- Qual [modo de compilação padrão de idioma](#) está definido.
- Se você estiver compilando a função em C ou C++.
- A opção de compilador [/eh](#) que você usa.
- Se a especificação de exceção for determinada explicitamente.

As especificações explícitas de exceção não são permitidas em funções C. Pressupõe-se que uma função C não gere exceções em /EHsce possa lançar exceções estruturadas em o/EHS, /EHA ou /EHAc.

A tabela a seguir resume se uma função C++ pode potencialmente ser lançada sob várias opções de manipulação de exceção de compilador:

FUNÇÃO	/EHSC	/EHS	/EHA	/EHAC
Função C++ sem especificação de exceção	Sim	Sim	Sim	Sim
Função C++ com <code>noexcept</code> <code>noexcept(true)</code> especificação de exceção, ou <code>throw()</code>	Não	Não	Sim	Sim
Função C++ com <code>noexcept(false)</code> <code>throw(...)</code> especificação de exceção, ou <code>throw(type)</code>	Sim	Sim	Sim	Sim

Exemplo

```

// exception_specification.cpp
// compile with: /EHs
#include <stdio.h>

void handler() {
    printf_s("in handler\n");
}

void f1(void) throw(int) {
    printf_s("About to throw 1\n");
    if (1)
        throw 1;
}

void f5(void) throw() {
    try {
        f1();
    }
    catch(...) {
        handler();
    }
}

// invalid, doesn't handle the int exception thrown from f1()
// void f3(void) throw() {
//     f1();
// }

void __declspec(nothrow) f2(void) {
    try {
        f1();
    }
    catch(int) {
        handler();
    }
}

// only valid if compiled without /EHc
// /EHc means assume extern "C" functions don't throw exceptions
extern "C" void f4(void);
void f4(void) {
    f1();
}

int main() {
    f2();

    try {
        f4();
    }
    catch(...) {
        printf_s("Caught exception from f4\n");
    }
    f5();
}

```

```

About to throw 1
in handler
About to throw 1
Caught exception from f4
About to throw 1
in handler

```

Confira também

[Instruções try, throw e catch \(C++\)](#)

[Práticas recomendadas do C++ moderno para exceções e tratamento de erros](#)

noexcept (C++)

02/09/2020 • 3 minutes to read • [Edit Online](#)

C++ 11: Especifica se uma função pode gerar exceções.

Sintaxe

noexcept-expressão:

`noexcept`

`noexcept (expressão de constante)`

parâmetros

expressão de constante

Uma expressão constante do tipo `bool` que representa se o conjunto de possíveis tipos de exceção está vazio. A versão incondicional é equivalente a `noexcept(true)`.

Comentários

Uma *expressão noexcept* é um tipo de *especificação de exceção*, um sufixo para uma declaração de função que representa um conjunto de tipos que podem ser correspondidos por um manipulador de exceção para qualquer exceção que saia de uma função. Operador condicional unário `noexcept(constant_expression)` em que *constant_expression* produz e `true` seu sinônimo incondicional `noexcept`, especifique que o conjunto de possíveis tipos de exceção que pode sair de uma função está vazio. Ou seja, a função nunca gera uma exceção e nunca permite que uma exceção seja propagada fora do seu escopo. O operador `noexcept(constant_expression)` onde *constant_expression* produz ou `false` a ausência de uma especificação de exceção (diferente de para um destruidor ou função de desalocação) indica que o conjunto de possíveis exceções que podem sair da função é o conjunto de todos os tipos.

Marque uma função `noexcept` somente se todas as funções que ele chama, direta ou indiretamente, também são `noexcept` ou `const`. O compilador não verifica necessariamente cada caminho de código em busca de exceções que possam surgir em uma `noexcept` função. Se uma exceção sair do escopo externo de uma função marcada `noexcept`, `std::Terminate` será invocado imediatamente e não haverá nenhuma garantia de que os destruidores de quaisquer objetos no escopo serão invocados. Use `noexcept` em vez do especificador de exceção dinâmica `throw()`, que agora é preferido no padrão. Recomendamos que você aplique `noexcept` a qualquer função que nunca permita que uma exceção propague a pilha de chamadas. Quando uma função é declarada `noexcept`, ela permite que o compilador gere um código mais eficiente em vários contextos diferentes. Para obter mais informações, consulte [especificações de exceção](#).

Exemplo

Uma função de modelo que copia seu argumento pode ser declarada `noexcept` na condição de que o objeto que está sendo copiado é um tipo de dados antigo (Pod). Essa função pode ser declarada da seguinte maneira:

```
#include <type_traits>

template <typename T>
T copy_object(const T& obj) noexcept(std::is_pod<T>)
{
    // ...
}
```

Confira também

[Práticas recomendadas do C++ moderno para exceções e tratamento de erros](#)

[Especificações de exceção \(throw, noexcept\)](#)

Exceções C++ não tratadas

02/09/2020 • 2 minutes to read • [Edit Online](#)

Se um manipulador correspondente (ou manipulador de reticências `catch`) não puder ser encontrado para a exceção atual, a `terminate` função de tempo de execução predefinida será chamada. (Você também pode chamar explicitamente `terminate` em qualquer um de seus manipuladores.) A ação padrão do `terminate` é chamar `abort`. Se você quiser que `terminate` chame outra função em seu programa antes de sair do aplicativo, chame a função `set_terminate` com o nome da função a ser chamada como seu único argumento. Você pode chamar `set_terminate` em qualquer momento do programa. A `terminate` rotina sempre chama a última função fornecida como um argumento para `set_terminate`.

Exemplo

O exemplo a seguir gera uma exceção `char *`, mas não contém um manipulador designado para capturar exceções de tipo `char *`. A chamada para `set_terminate` instrui `terminate` a chamar `term_func`.

```
// exceptions_Unhandled_Exceptions.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;
void term_func() {
    cout << "term_func was called by terminate." << endl;
    exit( -1 );
}
int main() {
    try
    {
        set_terminate( term_func );
        throw "Out of memory!" // No catch handler for this exception
    }
    catch( int )
    {
        cout << "Integer exception raised." << endl;
    }
    return 0;
}
```

Saída

```
term_func was called by terminate.
```

A função `term_func` deve encerrar o programa ou o thread atual, idealmente chamando `exit`. Caso contrário, retorna para seu chamador, `abort` é chamado.

Confira também

[Práticas recomendadas do C++ moderno para exceções e tratamento de erros](#)

Misturando exceções C (estruturadas) e C++

02/09/2020 • 3 minutes to read • [Edit Online](#)

Se você quiser escrever código portátil, o uso de SEH (manipulação de exceção estruturada) em um programa C++ não é recomendado. No entanto, às vezes você pode querer compilar usando [/EHa](#) e misturar exceções estruturadas e código-fonte C++ e precisa de algum recurso para lidar com os dois tipos de exceções. Como um manipulador de exceção estruturado não tem nenhum conceito de objetos ou exceções tipadas, ele não pode tratar exceções geradas pelo código C++. No entanto, os `catch` manipuladores C++ podem lidar com exceções estruturadas. A sintaxe de manipulação de exceção do C++ (`try`, `throw`, `catch`) não é aceita pelo compilador C, mas a sintaxe de manipulação de exceção estruturada (`__try`, `__except`, `__finally`) é suportada pelo compilador do C++.

Consulte [_set_se_translator](#) para obter informações sobre como tratar exceções estruturadas como exceções do C++.

Se você misturar exceções estruturadas e C++, esteja atento a esses possíveis problemas:

- Exceções de C++ e exceções estruturadas não podem ser misturadas na mesma função.
- Manipuladores de terminação (`__finally` blocos) são sempre executados, mesmo durante o desenrolamento após a geração de uma exceção.
- A manipulação de exceção do C++ pode capturar e preservar a semântica de desenrolamento em todos os módulos compilados com as [/EH](#) Opções do compilador, que habilitam a semântica de desenrolamento.
- Pode haver algumas situações em que as funções de destruidores não são chamadas para todos os objetos. Por exemplo, uma exceção estruturada pode ocorrer durante a tentativa de fazer uma chamada de função por meio de um ponteiro de função não inicializado. Se os parâmetros de função forem objetos construídos antes da chamada, os destruidores desses objetos não serão chamados durante o desenrolamento de pilha.

Próximas etapas

- [Usando o `setjmp` ou `longjmp` em programas em C++](#)

Veja mais informações sobre o uso de `setjmp` e `longjmp` em programas em C++.

- [Tratar exceções estruturadas no C++](#)

Veja exemplos de como você pode usar o C++ para lidar com exceções estruturadas.

Confira também

[Práticas recomendadas do C++ moderno para exceções e tratamento de erros](#)

Usando `setjmp` e `longjmp`

02/09/2020 • 2 minutes to read • [Edit Online](#)

Quando `setjmp` e `longjmp` são usados juntos, eles fornecem uma maneira de executar um não local `goto`.

Normalmente, eles são usados no código C para passar o controle de execução para tratamento de erros ou código de recuperação em uma rotina chamada anteriormente sem usar as convenções de chamada ou de retorno padrão.

Caution

Como `setjmp` e `longjmp` não oferecem suporte à destruição correta de objetos de quadro de pilha portamente entre compiladores C++, e como eles podem prejudicar o desempenho, impedindo a otimização em variáveis locais, não recomendamos seu uso em programas em C++. É recomendável usar `try` e `catch` construir em vez disso.

Se você decidir usar `setjmp` o e `longjmp` o em um programa C++, também incluirá `<setjmp.h>` ou `<setjmpex.h>` para garantir a interação correta entre as funções e a manipulação de exceção estruturada (SEH) ou o tratamento de exceções do C++.

Específico da Microsoft

Se você usar uma opção `/eh` para compilar código C++, os destruidores para objetos locais serão chamados durante o desenrolamento da pilha. No entanto, se você usar `/EHS` ou `/EHsc` para compilar e uma de suas funções que usa chamadas `noexcept` `longjmp`, o desenrolador de destruidor para essa função poderá não ocorrer, dependendo do estado do otimizador.

No código portátil, quando uma `longjmp` chamada é executada, a destruição correta de objetos baseados em quadros não é garantida explicitamente pelo padrão e pode não ter suporte de outros compiladores. Para informá-lo, no nível de aviso 4, uma chamada para `setjmp` causa aviso C4611: a interação entre '`_setjmp`' e a destruição de objeto do C++ não é portátil.

FINAL específico da Microsoft

Confira também

[Misturando exceções C \(estruturadas\) e C++](#)

Tratar exceções estruturadas no C++

02/09/2020 • 7 minutes to read • [Edit Online](#)

A principal diferença entre o tratamento de exceção estruturado (SEH) e a manipulação de exceção do C++ é que o modelo de manipulação de exceções do C++ lida em tipos, enquanto o modelo de manipulação de exceção estruturada C lida com exceções de um tipo; especificamente, `unsigned int`. Ou seja, as exceções de C são identificadas por um valor inteiro sem sinal, enquanto as exceções de C++ são identificadas pelo tipo de dados. Quando uma exceção estruturada é gerada em C, cada manipulador possível executa um filtro que examina o contexto de exceção C e determina se deve aceitar a exceção, passá-la para algum outro manipulador ou ignorá-la. Quando uma exceção é gerada em C++, ela pode ser de qualquer tipo.

Uma segunda diferença é que o modelo de manipulação de exceção estruturada C é conhecido como *assíncrono*, porque as exceções ocorrem secundários no fluxo de controle normal. O mecanismo de manipulação de exceção do C++ é totalmente *síncrono*, o que significa que as exceções ocorrem somente quando são geradas.

Quando você usa a opção de compilador `/EHs` ou `/EHsc`, nenhum manipulador de exceção C++ trata de exceções estruturadas. Essas exceções são tratadas apenas por `__except` manipuladores de exceção estruturados ou `__finally` manipuladores de terminação estruturados. Para obter informações, consulte [manipulação de exceção estruturada \(C/C++\)](#).

Na opção de compilador `/EHA`, se uma exceção C for gerada em um programa C++, ela poderá ser tratada por um manipulador de exceção estruturado com seu filtro associado ou por um manipulador de C++, o que `catch` for o mais próximo do contexto de exceção. Por exemplo, este programa C++ de exemplo gera uma exceção C dentro de um contexto de C++ `try` :

Exemplo – capturar uma exceção de C em um bloco catch do C++

```
// exceptions_Exception_Handling_Differences.cpp
// compile with: /EHa
#include <iostream>

using namespace std;
void SEHFunc( void );

int main() {
    try {
        SEHFunc();
    }
    catch( ... ) {
        cout << "Caught a C exception." << endl;
    }
}

void SEHFunc() {
    __try {
        int x, y = 0;
        x = 5 / y;
    }
    __finally {
        cout << "In finally." << endl;
    }
}
```

```
In finally.  
Caught a C exception.
```

Classes de wrapper de exceção C

Em um exemplo simples como o mostrado acima, a exceção C só pode ser capturada por uma elipse (...) `catch` cliques. Nenhuma informação sobre o tipo ou a natureza de exceção é comunicada ao manipulador. Embora esse método funcione, em alguns casos, talvez você queira definir uma transformação entre os dois modelos de manipulação de exceção para que cada exceção C seja associada a uma classe específica. Para transformar uma, você pode definir uma classe "wrapper" de exceção C, que pode ser usada ou derivada de para atribuir um tipo de classe específico a uma exceção de C. Ao fazer isso, cada exceção de C pode ser tratada separadamente por um manipulador de C++ específico `catch`, em vez de todos eles em um único manipulador.

Sua classe wrapper pode ter uma interface que consiste em algumas funções de membro que determinam o valor de exceção, e que acessam informações estendidas de contexto de exceção fornecidas pelo modelo da exceção de C. Você também pode querer definir um construtor padrão e um construtor que aceite um `unsigned int` argumento (para fornecer a representação de exceção de C subjacente) e um construtor de cópia bit-a-bit. Aqui está uma possível implementação de uma classe de wrapper de exceção C:

```
// exceptions_Exception_Handling_Differences2.cpp  
// compile with: /c  
class SE_Exception {  
private:  
    SE_Exception() {}  
    SE_Exception( SE_Exception& ) {}  
    unsigned int nSE;  
public:  
    SE_Exception( unsigned int n ) : nSE( n ) {}  
    ~SE_Exception() {}  
    unsigned int getSeNumber() {  
        return nSE;  
    }  
};
```

Para usar essa classe, instale uma função de tradução de exceção C personalizada que é chamada pelo mecanismo de manipulação de exceção interna sempre que uma exceção C é lançada. Dentro de sua função de conversão, você pode lançar qualquer exceção tipada (talvez um `SE_Exception` tipo ou um tipo de classe derivado de `SE_Exception`) que possa ser capturado por um manipulador de C++ correspondente apropriado `catch`. Em vez disso, a função translation pode retornar, o que indica que ela não trata a exceção. Se a própria função de conversão gerar uma exceção C, `Terminate` será chamado.

Para especificar uma função de tradução personalizada, chame a função `_set_se_translator` com o nome da função de tradução como seu único argumento. A função de tradução que você escreve é chamada uma vez para cada invocação de função na pilha que tem `try` blocos. Não há nenhuma função de tradução padrão; Se você não especificar um chamando `_set_se_translator`, a exceção C só poderá ser detectada por um manipulador de reticências `catch`.

Exemplo – usar uma função de tradução personalizada

Por exemplo, o código a seguir instala uma função personalizada de tradução e gerará a exceção de C que é envolvida pela classe `SE_Exception`:

```

// exceptions_Exception_Handling_Differences3.cpp
// compile with: /EHs
#include <stdio.h>
#include <eh.h>
#include <windows.h>

class SE_Exception {
private:
    SE_Exception() {}
    unsigned int nSE;
public:
    SE_Exception( SE_Exception& e ) : nSE(e.nSE) {}
    SE_Exception(unsigned int n) : nSE(n) {}
    ~SE_Exception() {}
    unsigned int getSeNumber() { return nSE; }
};

void SEFunc() {
    __try {
        int x, y = 0;
        x = 5 / y;
    }
    __finally {
        printf_s( "In finally\n" );
    }
}

void trans_func( unsigned int u, _EXCEPTION_POINTERS* pExp ) {
    printf_s( "In trans_func.\n" );
    throw SE_Exception( u );
}

int main() {
    _set_se_translator( trans_func );
    try {
        SEFunc();
    }
    catch( SE_Exception e ) {
        printf_s( "Caught a __try exception with SE_Exception.\n" );
        printf_s( "nSE = 0x%x\n", e.getSeNumber() );
    }
}

```

```

In trans_func.
In finally
Caught a __try exception with SE_Exception.
nSE = 0xc0000094

```

Confira também

[Combinação de exceções C \(estruturadas\) e C++](#)

Tratamento de exceções estruturado (C/C++)

02/09/2020 • 7 minutes to read • [Edit Online](#)

O SEH (manipulação de exceção estruturada) é uma extensão da Microsoft para C para lidar com determinadas situações de código excepcionais, como falhas de hardware, normalmente. Embora o Windows e o Microsoft C++ ofereçam suporte a SEH, recomendamos que você use a manipulação de exceção de C++ padrão ISO. Ele torna seu código mais portátil e flexível. No entanto, para manter o código existente ou para determinados tipos de programas, você ainda pode precisar usar SEH.

Específico da Microsoft:

Gramática

```
try-except-statement :  
    __try compound-statement __except ( expression ) compound-statement  
  
try-finally-statement :  
    __try compound-statement __finally compound-statement
```

Comentários

Com o SEH, você pode garantir que os recursos, como arquivos e blocos de memória, sejam liberados corretamente se a execução for encerrada inesperadamente. Você também pode lidar com problemas específicos — por exemplo, memória insuficiente — usando código estruturado conciso que não depende de `goto` instruções ou testes elaborados de códigos de retorno.

As `try-except` `try-finally` instruções e mencionadas neste artigo são extensões da Microsoft para a linguagem C. Elas oferecem suporte ao SEH permitindo que os aplicativos controlem um programa após os eventos que, caso contrário, finalizariam a execução. Ainda que o SEH funcione com arquivos de origem C++, ele não é projetado especificamente para C++. Se você usar o SEH em um programa C++ que você compilar usando a opção `/EHs` ou `/EHsc`, os destruidores para objetos locais serão chamados, mas outro comportamento de execução poderá não ser o esperado. Para obter uma ilustração, consulte o exemplo mais adiante neste artigo. Na maioria dos casos, em vez de SEH, recomendamos que você use a [manipulação de exceção de C++](#) padrão ISO, que o compilador do Microsoft C++ também suporta. Usando o tratamento de exceções C++, é possível garantir que o seu código seja mais portátil e tratar exceções de qualquer tipo.

Se você tiver código C que usa SEH, poderá misturá-lo com o código C++ que usa a manipulação de exceção do C++. Para obter informações, consulte [manipular exceções estruturadas em C++](#).

Existem dois mecanismos de SEH:

- [Manipuladores de exceção](#), ou `__except` blocos, que podem responder ou ignorar a exceção.
- [Manipuladores de encerramento](#) ou `__finally` blocos, que são sempre chamados, se uma exceção causa encerramento ou não.

Esses dois tipos de manipuladores são distintos, mas estão fortemente relacionados por meio de um processo conhecido como *desenrolar a pilha*. Quando ocorre uma exceção estruturada, o Windows procura o manipulador de exceção instalado mais recentemente que está ativo no momento. O manipulador pode executar uma de três ações:

- Não reconhecer a exceção e não passar o controle para outros manipuladores.

- Reconhecer a exceção mas ignorá-la.
- Confirmar a exceção e manipulá-la.

O manipulador de exceções que reconhece a exceção pode não estar na função em execução no momento da exceção. Ele pode estar em uma função muito maior na pilha. A função em execução no momento e quaisquer outras funções no quadro de pilhas são terminadas. Durante esse processo, a pilha é *rebobinada*. Ou seja, variáveis locais não estáticas de funções encerradas são limpas da pilha.

Como o sistema operacional desenrola a pilha, ele chama todos os manipuladores de término escritos para cada função. Usando um manipulador de encerramento, você limpa os recursos que, de outra forma, permanecerão abertos devido a um encerramento anormal. Se você tiver inserido uma seção crítica, poderá encerrá-la no manipulador de encerramento. Quando o programa for desligado, você poderá realizar outras tarefas de manutenção, como fechar e remover arquivos temporários.

Próximas etapas

- [Escrevendo um manipulador de exceção](#)
- [Escrevendo um manipulador de término](#)
- [Tratar exceções estruturadas no C++](#)

Exemplo

Como mencionado anteriormente, os destruidores para objetos locais são chamados se você usar o SEH em um programa C++ e compilá-lo usando a `/EHs` `/EHsc` opção ou. No entanto, o comportamento durante a execução pode não ser o esperado se você também estiver usando exceções C++. Este exemplo demonstra essas diferenças comportamentais.

```

#include <stdio.h>
#include <Windows.h>
#include <exception>

class TestClass
{
public:
    ~TestClass()
    {
        printf("Destroying TestClass!\r\n");
    }
};

__declspec(noinline) void TestCPPEX()
{
#ifdef CPPEX
    printf("Throwing C++ exception\r\n");
    throw std::exception("");
#else
    printf("Triggering SEH exception\r\n");
    volatile int *pInt = 0x00000000;
    *pInt = 20;
#endif
}

__declspec(noinline) void TestExceptions()
{
    TestClass d;
    TestCPPEX();
}

int main()
{
    __try
    {
        TestExceptions();
    }
    __except(EXCEPTION_EXECUTE_HANDLER)
    {
        printf("Executing SEH __except block\r\n");
    }

    return 0;
}

```

Se você usar `/EHsc` o para compilar esse código, mas a macro de controle de teste local `CPPEX` estiver indefinida, o `TestClass` destruidor não será executado. A saída se parece com esta:

```

Triggering SEH exception
Executing SEH __except block

```

Se você usar `/EHsc` o para compilar o código e `CPPEX` for definido usando `/DCPPEX` (para que uma exceção de C++ seja lançada), o `TestClass` destruidor será executado e a saída terá esta aparência:

```

Throwing C++ exception
Destroying TestClass!
Executing SEH __except block

```

Se você usar `/EHs` o para compilar o código, o `TestClass` destruidor executará se a exceção foi gerada usando `std::throw` ou usando Seh para disparar a exceção. Ou seja, se `CPPEX` é definido ou não. A saída se parece com esta:

```
Throwing C++ exception
Destroying TestClass!
Executing SEH __except block
```

Para obter mais informações, consulte [/EH](#) (modelo de tratamento de exceção).

ENCERRAR específico da Microsoft

Confira também

[Tratamento de exceção](#)

[Palavras-chave](#)

[`<exception>`](#)

[Erros e tratamento de exceção](#)

[Manipulação de exceção estruturada \(Windows\)](#)

Escrevendo um manipulador de exceção

25/03/2020 • 2 minutes to read • [Edit Online](#)

Normalmente, os manipuladores de exceções são usados para responder a erros específicos. Você pode usar a sintaxe de manipulação de exceções para filtrar todas as exceções que não sejam aquelas que você sabe tratar. Outras exceções devem ser passadas para outros manipuladores (possivelmente na biblioteca em tempo de execução ou no sistema operacional) criados para procurar essas exceções específicas.

Os manipuladores de exceções usam a instrução try-except.

Que mais você deseja saber?

- [A instrução try-Except](#)
- [Escrevendo um filtro de exceção](#)
- [Gerando exceções de software](#)
- [Exceções de hardware](#)
- [Restrições em manipuladores de exceção](#)

Confira também

[Tratamento de exceções estruturado \(C/C++\)](#)

Instrução `try-except`

02/09/2020 • 9 minutes to read • [Edit Online](#)

A `try-except` instrução é uma extensão **específica da Microsoft** que dá suporte à manipulação de exceção estruturada nas linguagens C e C++.

```
// . . .
__try {
    // guarded code
}
__except ( /* filter expression */ ) {
    // termination code
}
// . . .
```

Gramática

```
try-except-statement :
    __try compound-statement __except ( expression ) compound-statement
```

Comentários

A `try-except` instrução é uma extensão da Microsoft para as linguagens C e C++. Ele permite que os aplicativos de destino tenham controle quando ocorrem eventos que normalmente encerram a execução do programa. Esses eventos são chamados de *exceções estruturadas* ou *exceções* de forma abreviada. O mecanismo que lida com essas exceções é chamado de SEH (*manipulação de exceção estruturada*).

Para obter informações relacionadas, consulte a [instrução try-finally](#).

As exceções podem ser baseadas em hardware ou em software. A manipulação de exceção estruturada é útil mesmo quando os aplicativos não podem se recuperar completamente de exceções de hardware ou software. O SEH torna possível exibir informações de erro e interceptar o estado interno do aplicativo para ajudar a diagnosticar o problema. Ele é especialmente útil para problemas intermitentes que não são fáceis de reproduzir.

NOTE

A manipulação de exceção estruturada funciona com Win32 para arquivos de código-fonte em C e C++. No entanto, ele não foi projetado especificamente para C++. Você pode garantir que o código seja mais portátil usando a manipulação de exceção de C++. Além disso, a manipulação de exceção de C++ é mais flexível, pois pode tratar exceções de qualquer tipo. Para programas em C++, recomendamos que você use as instruções de manipulação de exceção C++ nativas: [try](#), [catch](#) e [throw](#).

A instrução composta após a `__try` cláusula é o *corpo* ou a seção *protégida*. A `__except` expressão também é conhecida como a expressão de *filtro*. Seu valor determina como a exceção é tratada. A instrução composta após a `__except` cláusula é o manipulador de exceção. O manipulador Especifica as ações a serem executadas se uma exceção for gerada durante a execução da seção Body. A execução procede da seguinte maneira:

1. A seção protegida é executada.
2. Se nenhuma exceção ocorrer durante a execução da seção protegida, a execução continuará na instrução

após a `__except` cláusula.

3. Se ocorrer uma exceção durante a execução da seção protegida ou, em qualquer rotina, a seção protegida chamar, a `__except` expressão será avaliada. Há três valores possíveis:

- `EXCEPTION_CONTINUE_EXECUTION` (-1) Exceção descartada. Continue a execução no ponto onde ocorreu a exceção.
- `EXCEPTION_CONTINUE_SEARCH` (0) a exceção não é reconhecida. Continue pesquisando a pilha para obter um manipulador, primeiro para as instruções de contenção `try-except`, em seguida, para os manipuladores com a próxima precedência mais alta.
- `EXCEPTION_EXECUTE_HANDLER` (1) a exceção é reconhecida. Transfira o controle para o manipulador de exceção executando a `__except` instrução composta e continue a execução após o `__except` bloco.

A `__except` expressão é avaliada como uma expressão C. Ele é limitado a um único valor, ao operador de expressão condicional ou ao operador de vírgula. Se um processamento mais extenso for necessário, a expressão poderá chamar uma rotina que retorne um dos três valores listados acima.

Cada aplicativo pode ter seu próprio manipulador de exceção.

Não é válido pular para uma `try` instrução, mas é válido para saltar de uma. O manipulador de exceção não será chamado se um processo for encerrado no meio da execução de uma `try-except` instrução.

Para compatibilidade com versões anteriores, `_try`, `_except` e `_leave` são sinônimos para `try`, `except` e `leave` a menos que a opção do compilador [/za \(desabilitar extensões de idioma\)](#) seja especificada.

A `__leave` palavra-chave

A `__leave` palavra-chave é válida somente na seção protegida de uma `try-except` instrução e seu efeito é saltar para o final da seção protegida. A execução continua na primeira instrução após o manipulador de exceção.

Uma `goto` instrução também pode sair da seção protegida e não degrada o desempenho como faz em uma instrução `try-finally`. Isso porque o desenrolamento de pilha não ocorre. No entanto, recomendamos que você use a `__leave` palavra-chave em vez de uma `goto` instrução. O motivo é que você tem menos probabilidade de fazer um erro de programação se a seção protegida for grande ou complexa.

Funções intrínsecas de manipulação de exceções estruturadas

A manipulação de exceção estruturada fornece duas funções intrínsecas que estão disponíveis para uso com a `try-except` instrução: [GetExceptionCode](#) e [GetExceptionInformation](#).

`GetExceptionCode` Retorna o código (um número inteiro de 32 bits) da exceção.

A função intrínseca `GetExceptionInformation` retorna um ponteiro para uma estrutura de `EXCEPTION_POINTERS` que contém informações adicionais sobre a exceção. Por esse ponteiro, você pode acessar qual era o estado do computador no momento em que ocorreu uma exceção de hardware. A estrutura é a seguinte:

```
typedef struct _EXCEPTION_POINTERS {
    PEXCEPTION_RECORD ExceptionRecord;
    PCONTEXT ContextRecord;
} EXCEPTION_POINTERS, *PEXCEPTION_POINTERS;
```

Os tipos de ponteiro `PEXCEPTION_RECORD` e `PCONTEXT` são definidos no arquivo de inclusão e `<winnt.h>`
`_EXCEPTION_RECORD` e `_CONTEXT` são definidos no arquivo de inclusão `<excpt.h>`

Você pode usar `GetExceptionCode` dentro do manipulador de exceção. No entanto, você pode usar `GetExceptionInformation` somente dentro da expressão de filtro de exceção. As informações apontadas são geralmente na pilha e não estão mais disponíveis quando o controle é transferido para o manipulador de exceção.

A função intrínseca `AbnormalTermination` está disponível dentro de um manipulador de encerramento. Retornará 0 se o corpo da instrução `try-finally` terminar em sequência. Em todos os outros casos, retorna 1.

`<excpt.h>` define alguns nomes alternativos para esses intrínsecos:

`GetExceptionCode` equivale a `_exception_code`

`GetExceptionInformation` equivale a `_exception_info`

`AbnormalTermination` equivale a `_abnormal_termination`

Exemplo

```
// exceptions_try_except_Statement.cpp
// Example of try-except and try-finally statements
#include <stdio.h>
#include <windows.h> // for EXCEPTION_ACCESS_VIOLATION
#include <excpt.h>

int filter(unsigned int code, struct _EXCEPTION_POINTERS *ep)
{
    puts("in filter.");
    if (code == EXCEPTION_ACCESS_VIOLATION)
    {
        puts("caught AV as expected.");
        return EXCEPTION_EXECUTE_HANDLER;
    }
    else
    {
        puts("didn't catch AV, unexpected.");
        return EXCEPTION_CONTINUE_SEARCH;
    };
}

int main()
{
    int* p = 0x00000000;    // pointer to NULL
    puts("hello");
    __try
    {
        puts("in try");
        __try
        {
            puts("in try");
            *p = 13;    // causes an access violation exception;
        }
        __finally
        {
            puts("in finally. termination: ");
            puts(AbnormalTermination() ? "\tabnormal" : "\tnormal");
        }
    }
    __except(filter(GetExceptionCode(), GetExceptionInformation()))
    {
        puts("in except");
    }
    puts("world");
}
```

Saída

```
hello
in try
in try
in filter.
caught AV as expected.
in finally. termination:
    abnormal
in except
world
```

Confira também

[Escrevendo um manipulador de exceção](#)

[Manipulação de exceção estruturada \(C/C++\)](#)

[Palavras-chave](#)

Escrevendo um filtro de exceção

15/04/2020 • 4 minutes to read • [Edit Online](#)

Você pode lidar com uma exceção indo diretamente ao nível do manipulador de exceção ou continuando a execução. Em vez de usar o código do manipulador de exceção para lidar com a exceção e cair, você pode usar o *filtro* para limpar o problema e, em seguida, retornando -1, retomar o fluxo normal sem limpar a pilha.

NOTE

Algumas exceções não podem ser retomadas. Se o *filtro* avaliar para -1 para tal exceção, o sistema aumentará uma nova exceção. Quando você chama `RaiseException`, você determina se a exceção continuará.

Por exemplo, o código a seguir usa uma chamada de função na expressão *do filtro*: esta função lida com o problema e, em seguida, retorna -1 para retomar o fluxo normal de controle:

```
// exceptions_Writing_an_Exception_Filter.cpp
#include <windows.h>
int main() {
    int Eval_Exception( int );

    __try {}
    __except ( Eval_Exception( GetExceptionCode( ) ) ) {
        ;
    }

    void ResetVars( int ) {}
    int Eval_Exception ( int n_except ) {
        if ( n_except != STATUS_INTEGER_OVERFLOW &&
            n_except != STATUS_FLOAT_OVERFLOW ) // Pass on most exceptions
        return EXCEPTION_CONTINUE_SEARCH;

        // Execute some code to clean up problem
        ResetVars( 0 ); // initializes data to 0
        return EXCEPTION_CONTINUE_EXECUTION;
    }
}
```

É uma boa ideia usar uma chamada de função na expressão do *filtro* sempre que o *filtro* precisa fazer algo complexo. Avaliar a expressão causa a execução da função, nesse caso, `Eval_Exception`.

Observe o uso de `GetExceptionCode` para determinar a exceção. Você deve chamar essa função dentro do próprio filtro. `Eval_Exception` não `GetExceptionCode` pode chamar, mas deve ter o código de exceção passado para ele.

Esse manipulador passa o controle para outro manipulador, a menos que a exceção seja um inteiro ou um estouro de ponto flutuante. Se for o caso, o manipulador chamará uma função (`ResetVars` é apenas um exemplo, não uma função de API) para redefinir alguns variáveis globais. *O statement-block-2*, que neste exemplo está vazio, nunca pode ser executado porque `Eval_Exception` nunca retorna `EXCEPTION_EXECUTE_HANDLER` (1).

Usar uma chamada de função é uma boa técnica de uso geral para lidar com expressões de filtro complexas.

Outros dois recursos da linguagem C úteis são:

- O operador condicional
- O operador vírgula

O operador condicional geralmente é útil, pois pode ser usado para verificar se há um código de retorno específico e retornar um de dois valores diferentes. Por exemplo, o filtro no código a seguir reconhece a exceção somente se a exceção for STATUS_INTEGER_OVERFLOW:

```
__except( GetExceptionCode() == STATUS_INTEGER_OVERFLOW ? 1 : 0 ) {
```

O propósito do operador condicional nesse caso é basicamente fornecer clareza, pois o código a seguir gera os mesmos resultados:

```
__except( GetExceptionCode() == STATUS_INTEGER_OVERFLOW ) {
```

O operador condicional é mais útil em situações em que você pode querer que o filtro avalie para -1, EXCEPTION_CONTINUE_EXECUTION.

O operador vírgula permite executar várias operações independentes dentro de uma única expressão. O efeito é basicamente o de executar várias instruções e depois retornar o valor da última expressão. Por exemplo, o código a seguir armazena o código de exceção em uma variável e o testa:

```
__except( nCode = GetExceptionCode(), nCode == STATUS_INTEGER_OVERFLOW )
```

Confira também

[Escrevendo um manipulador de exceção](#)

[Tratamento estruturado de exceções \(C/C++\)](#)

Acionando exceções de software

25/03/2020 • 4 minutes to read • [Edit Online](#)

Algumas das origens mais comuns de erros do programa não são sinalizadas como exceções pelo sistema. Por exemplo, se você tenta alocar um bloco de memória, mas não há memória suficiente, o tempo de execução ou a função de API não geram uma exceção, mas retornam um código de erro.

No entanto, você pode tratar qualquer condição como uma exceção detectando essa condição em seu código e, em seguida, relatando-a chamando a função `RaiseException`. Ao sinalizar erros dessa maneira, você aproveita as vantagens de manipulação de exceções estruturada em qualquer tipo de erro de tempo de execução.

Para usar a manipulação de exceção estruturada com erros:

- Defina seu próprio código de exceção para o evento.
- Chame `RaiseException` ao detectar um problema.
- Use filtros de manipulação de exceções para testar o código de exceção definido.

O arquivo `<Winerror.h>` mostra o formato dos códigos de exceção. Para verificar se você não definiu um código em conflito com um código de exceção existente, defina o terceiro bit mais significativo como 1. Os quatro bit mais significativos devem ser definidos como mostrado na tabela a seguir.

BITS	CONFIGURAÇÃO BINÁRIA RECOMENDADA	DESCRIÇÃO
31-30	11	Esses dois bits descrevem o status básico de código: 11 = erro, 00 = êxito, 01 = informativo, 10 = aviso.
29	1	Bit cliente. Definido como 1 para códigos definidos pelo usuário.
28	0	Bit reservado. (Deixe definido como 0.)

Você pode definir os dois primeiros bits com uma configuração diferente do 11 binário se você desejar, embora a configuração de "erro" seja apropriada para a maioria das exceções. É importante lembrar de definir os bits 29 e 28 conforme mostrado na tabela anterior.

O código de erro resultante, portanto, deve ter os quatro bits mais altos definidos como hexadecimal E. Por exemplo, as definições a seguir definem códigos de exceção que não entram em conflito com nenhum código de exceção do Windows. (No entanto, talvez seja necessário verificar se os códigos são usados por DLL de terceiros.)

```
#define STATUS_INSUFFICIENT_MEM      0xE0000001
#define STATUS_FILE_BAD_FORMAT        0xE0000002
```

Depois que você tiver definido um código de exceção, poderá usá-lo para gerar uma exceção. Por exemplo, o código a seguir gera a exceção `STATUS_INSUFFICIENT_MEM` em resposta a um problema de alocação de memória:

```
lpstr = _malloc( nBufferSize );
if (lpstr == NULL)
    RaiseException( STATUS_INSUFFICIENT_MEM, 0, 0, 0);
```

Se você quiser simplesmente gerar uma exceção, pode definir os últimos três parâmetros como 0. Os últimos três parâmetros são úteis para passar informações adicionais e definir um sinalizador que impeça manipuladores de continuarem a execução. Consulte a função [RaiseException](#) no SDK do Windows para obter mais informações.

Em seus filtros de manipulação de exceções, você pode testar os códigos que você definiu. Por exemplo:

```
__try {  
    ...  
}  
__except (GetExceptionCode() == STATUS_INSUFFICIENT_MEM ||  
         GetExceptionCode() == STATUS_FILE_BAD_FORMAT )
```

Confira também

[Escrevendo um manipulador de exceção](#)

[Manipulação de exceção estruturada \(C++C/\)](#)

Exceções de hardware

25/03/2020 • 3 minutes to read • [Edit Online](#)

A maioria das exceções padrão reconhecidas pelo sistema operacional são exceções definidas por hardware. O Windows reconhece algumas exceções de software de nível baixo, mas elas geralmente são mais bem tratadas pelo sistema operacional.

O Windows mapeia os erros de hardware de processadores diferentes para os códigos de exceção nesta seção. Em alguns casos, um processador pode gerar somente um subconjunto dessas exceções. O Windows pré-processa informações sobre a exceção e emite o código de exceção apropriado.

As exceções de hardware reconhecidas pelo windows são resumidas na seguinte tabela:

CÓDIGO DA EXCEÇÃO	CAUSA DA EXCEÇÃO
STATUS_ACCESS_VIOLATION	Leitura ou gravação em um local de memória inacessível.
STATUS_BREAKPOINT	Encontro de um ponto de interrupção definido por hardware; usado somente por depuradores.
STATUS_DATATYPE_MISALIGNMENT	Leitura ou gravação de dados em um endereço que não está alinhado corretamente; por exemplo, as entidades de 16 bits devem ser alinhadas em limites de 2 bytes. (Não aplicável a processadores Intel 80x86.)
STATUS_FLOAT_DIVIDE_BY_ZERO	Divisão do tipo de ponto flutuante por 0,0.
STATUS_FLOAT_OVERFLOW	Expoente positivo máximo do tipo de ponto flutuante excedido.
STATUS_FLOAT_UNDERFLOW	Magnitude excedida do expoente negativo mais baixo do tipo de ponto flutuante.
STATUS_FLOATING_RESEVERED_OPERAND	Usando um formato de ponto flutuante reservado (uso inválido de formato).
STATUS_ILLEGAL_INSTRUCTION	Tentativa de executar um código de instrução não definido pelo processador.
STATUS_PRIVILEGED_INSTRUCTION	Execução de uma instrução não permitida no modo atual do computador.
STATUS_INTEGER_DIVIDE_BY_ZERO	Divisão de um tipo de inteiro por 0.
STATUS_INTEGER_OVERFLOW	Tentativa de uma operação que excede o intervalo do inteiro.
STATUS_SINGLE_STEP	Execução de uma instrução no modo de etapa única; usado somente por depuradores.

Muitas das exceções listadas na tabela anterior devem ser tratadas por depuradores, pelo sistema operacional, ou outro código de nível baixo. Com a exceção de erros de inteiro e de ponto flutuante, seu código não deve tratar esses erros. Assim, geralmente você deve usar o filtro de tratamento de exceções para ignorar exceções (avaliado

como 0). Caso contrário, você poderá impedir que os mecanismos de nível inferior respondam adequadamente. No entanto, você pode tomar as precauções apropriadas contra o efeito potencial desses erros de nível inferior [escrevendo manipuladores de encerramento](#).

Confira também

[Escrevendo um manipulador de exceção](#)

[Tratamento de exceções estruturado \(C/C++\)](#)

Restrições em manipuladores de exceção

02/09/2020 • 2 minutes to read • [Edit Online](#)

A principal limitação do uso de manipuladores de exceção no código é que você não pode usar uma `goto` instrução para saltar para um `__try` bloco de instruções. Em vez disso, você deve digitar o bloco de instruções por meio do fluxo de controle normal. Você pode saltar para fora de um `__try` bloco de instrução e pode aninhar manipuladores de exceção conforme escolher.

Confira também

[Escrevendo um manipulador de exceção](#)

[Manipulação de exceção estruturada \(C/C++\)](#)

Escrevendo um manipulador de término

25/03/2020 • 2 minutes to read • [Edit Online](#)

Diferente de um manipulador de exceção, um manipulador de término sempre é executado, independentemente do bloco de código protegido encerrado normalmente. O único propósito do manipulador de término deve ser garantir que os recursos, como memória, identificadores e arquivos, sejam fechados corretamente, independentemente de como uma seção de código termina a execução.

Os manipuladores de término usam a instrução try-finally.

Que mais você deseja saber?

- [A instrução try – finally](#)
- [Limpando recursos](#)
- [Tempo de ações na manipulação de exceção](#)
- [Restrições em manipuladores de encerramento](#)

Confira também

[Tratamento de exceções estruturado \(C/C++\)](#)

Instrução `try-finally`

02/09/2020 • 8 minutes to read • [Edit Online](#)

A `try-finally` instrução é uma extensão **específica da Microsoft** que dá suporte à manipulação de exceção estruturada nas linguagens C e C++.

Sintaxe

A sintaxe a seguir descreve a `try-finally` instrução:

```
// . . .
__try {
    // guarded code
}
__finally {
    // termination code
}
// . . .
```

Gramática

```
try-finally-statement :
    __try compound-statement __finally compound-statement
```

A `try-finally` instrução é uma extensão da Microsoft para as linguagens C e C++ que permitem que os aplicativos de destino garantam a execução do código de limpeza quando a execução de um bloco de código é interrompida. A limpeza consiste em tarefas como desalocar memória, fechar arquivos e liberar identificadores de arquivos. A instrução `try-finally` é especialmente útil para rotinas que têm vários locais onde uma verificação é feita para um erro que pode causar o retorno prematuro da rotina.

Para obter informações relacionadas e um exemplo de código, consulte [try-except instrução](#). Para obter mais informações sobre manipulação de exceção estruturada em geral, consulte [manipulação de exceção estruturada](#). Para obter mais informações sobre como lidar com exceções em aplicativos gerenciados com C++/CLI, consulte [tratamento de exceção em /clr](#).

NOTE

A manipulação de exceção estruturada funciona com Win32 para arquivos de código-fonte em C e C++. No entanto, não é projetada especificamente para C++. Você pode garantir que o código seja mais portátil usando a manipulação de exceção de C++. Além disso, a manipulação de exceção de C++ é mais flexível, pois pode tratar exceções de qualquer tipo. Para programas em C++, é recomendável que você use o mecanismo de manipulação de exceção do C++ (instruções `try`, `catch` e `throw`).

A instrução composta após a `__try` cláusula é a seção protegida. A instrução composta após a `__finally` cláusula é o manipulador de terminação. O manipulador Especifica um conjunto de ações que são executadas quando a seção protegida é encerrada, se ela sai da seção protegida por uma exceção (encerramento anormal) ou, por padrão, se enquadra (término normal).

O controle alcança uma `__try` instrução por execução sequencial simples (que se enquadra). Quando o controle

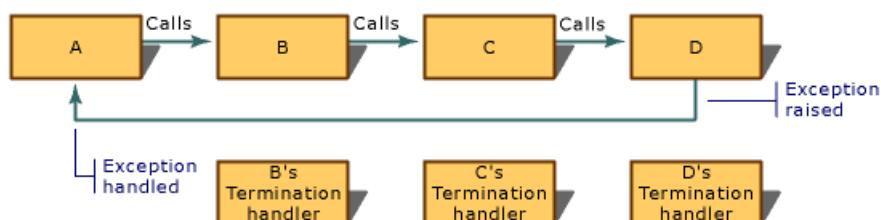
entra no `__try`, seu manipulador associado se torna ativo. Se o fluxo de controle chegar ao fim do bloco `try`, a execução continuará da seguinte maneira:

1. O manipulador de término é invocado.
2. Quando o manipulador de encerramento for concluído, a execução continuará após a `__finally` instrução. No entanto, a seção protegida termina (por exemplo, por meio `goto` de um corpo protegido ou uma `return` instrução), o manipulador de terminação é executado *antes* do fluxo de controle ser movido para fora da seção protegida.

Uma `__finally` instrução não bloqueia a pesquisa de um manipulador de exceção apropriado.

Se ocorrer uma exceção no `__try` bloco, o sistema operacional deverá encontrar um manipulador para a exceção ou o programa falhará. Se um manipulador for encontrado, todos os blocos e todos `__finally` serão executados e a execução será retomada no manipulador.

Por exemplo, imagine que uma série de chamadas de função vincula a função A à função D, conforme mostrado na figura a seguir. Cada função tem um manipulador de encerramento. Se uma exceção é gerada na função D e tratada na A, os manipuladores de encerramento são chamados nessa ordem à medida que o sistema desenrola a pilha: D, C, B.



Ordem de execução do manipulador de encerramento

NOTE

O comportamento de `try - finally` é diferente de algumas outras linguagens que dão suporte ao uso do `finally`, como o C#. Um único `__try` pode ter um, mas não ambos, de `__finally` e `__except`. Se ambos devem ser usados juntos, uma instrução `try-except` externa deve incluir a instrução interna `try-finally`. As regras que especificam quando cada bloco é executado também são diferentes.

Para compatibilidade com versões anteriores, `__try`, `__finally` e `__leave` são sinônimos para `__try`, `__finally` e `__leave` a menos que a opção do compilador [/za](#) (desabilitar extensões de linguagem) seja especificada.

A palavra-chave `__leave`

A `__leave` palavra-chave é válida somente na seção protegida de uma `try-finally` instrução e seu efeito é saltar para o final da seção protegida. A execução continua na primeira instrução do manipulador de encerramento.

Uma `goto` instrução também pode sair da seção protegida, mas ela degrada o desempenho porque ele invoca o desenrolamento de pilha. A `__leave` instrução é mais eficiente porque não causa o desenrolamento da pilha.

encerramento anormal

Sair de uma `try-finally` instrução usando a função de tempo de execução `longjmp` é considerado encerramento anormal. Não é legal pular para uma `__try` instrução, mas é legal pular de uma. Todas as `__finally` instruções que estão ativas entre o ponto de saída (término normal do `__try` bloco) e o destino (o `__except` bloco que manipula a exceção) devem ser executados. Ele é chamado de *desenrolamento local*.

Se um `__try` bloco for encerrado prematuramente por qualquer motivo, incluindo um salto para fora do bloco,

o sistema executará o `__finally` bloco associado como parte do processo de desenrolar a pilha. Nesses casos, a `AbnormalTermination` função retorna `true` se for chamada de dentro do `__finally` bloco; caso contrário, retornará `false`.

O manipulador de encerramento não será chamado se um processo for eliminado no meio da execução de uma `try-finally` instrução.

ENCERRAR específico da Microsoft

Confira também

[Escrevendo um manipulador de término](#)

[Manipulação de exceção estruturada \(C/C++\)](#)

[Palavras-chave](#)

[Sintaxe do manipulador de terminação](#)

Limpando recursos

02/09/2020 • 2 minutes to read • [Edit Online](#)

Durante a execução do manipulador de encerramento, talvez você não saiba quais recursos foram adquiridos antes de o manipulador de encerramento ser chamado. É possível que o `__try` bloco de instruções tenha sido interrompido antes de todos os recursos serem adquiridos, para que nem todos os recursos fossem abertos.

Para ser seguro, você deve verificar para ver quais recursos estão abertos antes de prosseguir com a limpeza do tratamento de encerramento. Um procedimento recomendado é fazer o seguinte:

1. Inicialize os identificadores como NULL.
2. No `__try` bloco de instruções, adquira recursos. Os identificadores são definidos como valores positivos conforme o recurso é adquirido.
3. No `__finally` bloco de instruções, libere cada recurso cuja variável identificador ou sinalizador correspondente seja diferente de zero ou não seja nulo.

Exemplo

Por exemplo, o código a seguir usa um manipulador de encerramento para fechar três arquivos e liberar um bloco de memória. Esses recursos foram adquiridos no `__try` bloco de instruções. Antes de limpar um recurso, o código verifica primeiro se o recurso foi adquirido.

```
// exceptions_Cleaning_up_Resources.cpp
#include <stdlib.h>
#include <malloc.h>
#include <stdio.h>
#include <windows.h>

void fileOps() {
    FILE *fp1 = NULL,
        *fp2 = NULL,
        *fp3 = NULL;
    LPVOID lpvoid = NULL;
    errno_t err;

    __try {
        lpvoid = malloc( BUFSIZ );

        err = fopen_s(&fp1, "ADDRESS.DAT", "w+" );
        err = fopen_s(&fp2, "NAMES.DAT", "w+" );
        err = fopen_s(&fp3, "CARS.DAT", "w+" );
    }

    __finally {
        if ( fp1 )
            fclose( fp1 );
        if ( fp2 )
            fclose( fp2 );
        if ( fp3 )
            fclose( fp3 );
        if ( lpvoid )
            free( lpvoid );
    }
}

int main() {
    fileOps();
}
```

Confira também

[Escrevendo um manipulador de término](#)

[Manipulação de exceção estruturada \(C/C++\)](#)

Tempo de tratamento de exceções: Um resumo

02/09/2020 • 3 minutes to read • [Edit Online](#)

Um manipulador de encerramento é executado, independentemente de como o `__try` bloco de instrução é encerrado. As causas incluem sair do `__try` bloco, uma `longjmp` instrução que transfere o controle do bloco e desenrolando a pilha devido à manipulação de exceção.

NOTE

O compilador do Microsoft C++ dá suporte a duas formas das `setjmp` `longjmp` instruções e. A versão rápida ignora a manipulação de término, mas é mais eficiente. Para usar essa versão, inclua o arquivo `<setjmp.h>`. A outra versão oferece suporte à manipulação de término conforme descrito no parágrafo anterior. Para usar essa versão, inclua o arquivo `<setjmpex.h>`. O aumento no desempenho da versão rápida depende da configuração de hardware.

O sistema operacional executa todos os manipuladores de término na ordem apropriada antes que qualquer outro código possa ser executado, incluindo o corpo de um manipulador de exceção.

Quando a causa da interrupção é uma exceção, o sistema deve primeiro executar a parte de filtro de um ou mais manipuladores de exceções antes de decidir o que terminar. A ordem de eventos é:

1. Uma exceção é acionada.
2. O sistema examina a hierarquia de manipuladores de exceção ativa e executa o filtro do manipulador com precedência mais alta. Esse é o manipulador de exceção instalado mais recentemente e mais profundamente aninhado, indo por blocos e chamadas de função.
3. Se esse filtro passar o controle (retorna 0), o processo continuará até encontrar um filtro que não passe o controle.
4. Se esse filtro retornar -1, a execução continuará onde a exceção foi gerada e nenhum encerramento ocorrerá.
5. Se o filtro retornar 1, ocorrerão os seguintes eventos:
 - O sistema desenrola a pilha: limpa todos os quadros de pilha entre onde a exceção foi gerada e o quadro de pilha que contém o manipulador de exceção.
 - À medida que a pilha é desenrolada, cada manipulador de término na pilha é executado.
 - O próprio manipulador de exceção é executado.
 - O controle passa para a linha de código após o final desse manipulador de exceção.

Confira também

[Escrevendo um manipulador de término](#)
[Manipulação de exceção estruturada \(C/C++\)](#)

Restrições em manipuladores de término

02/09/2020 • 2 minutes to read • [Edit Online](#)

Você não pode usar uma `goto` instrução para saltar para `__try` um bloco de instruções ou um bloco de `__finally` instruções. Em vez disso, você deve digitar o bloco de instruções por meio do fluxo de controle normal. (No entanto, você pode saltar de um `__try` bloco de instrução.) Além disso, você não pode aninhar um manipulador de exceção ou manipulador de terminação dentro de um `__finally` bloco.

Alguns tipos de código permitidos em um manipulador de encerramento produzem resultados questionáveis, portanto, você deve usá-los com cuidado, se houver. Uma é uma `goto` instrução que sai de um `__finally` bloco de instrução. Se o bloco for executado como parte do encerramento normal, nada acontecerá incomum. Mas se o sistema estiver desenrolando a pilha, isso interromperá. Em seguida, a função atual obtém o controle como se não houvesse uma finalização anormal.

Uma `return` instrução dentro de um `__finally` bloco de instruções apresenta aproximadamente a mesma situação. O controle retorna para o chamador imediato da função que contém o manipulador de encerramento. Se o sistema estiver desenrolando a pilha, esse processo será interrompido. Em seguida, o programa prossegue como se nenhuma exceção tivesse sido gerada.

Confira também

[Escrevendo um manipulador de término](#)

[Manipulação de exceção estruturada \(C/C++\)](#)

Transportando exceções entre threads

02/09/2020 • 20 minutes to read • [Edit Online](#)

O compilador do Microsoft C++ (MSVC) dá suporte à *transportação de uma exceção* de um thread para outro. O transporte de exceções permite que você capture uma exceção em um thread e faça parecer que ela tenha sido lançada em outro thread. Você pode usar esse recurso, por exemplo, para gravar um aplicativo multi-threaded em que o thread primário trata todas as exceções lançadas por seus threads secundários. O transporte de exceções é útil para a maioria dos desenvolvedores que cria bibliotecas ou sistemas de programação paralela. Para implementar exceções de transporte, o MSVC fornece o tipo de `exception_ptr` e as funções `current_exception`, `rethrow_exception` e `make_exception_ptr`.

Sintaxe

```
namespace std
{
    typedef unspecified exception_ptr;
    exception_ptr current_exception();
    void rethrow_exception(exception_ptr p);
    template<class E>
        exception_ptr make_exception_ptr(E e) noexcept;
}
```

Parâmetros

não especificado

Uma classe interna não especificada que é usada para implementar o tipo `exception_ptr`.

DTI

Um objeto `exception_ptr` que faz referência a uma exceção.

Oriental

Uma classe que representa uma exceção.

Oriental

Uma instância da classe `E` do parâmetro.

Valor retornado

A função `current_exception` retorna um objeto `exception_ptr` que faz referência à exceção que está atualmente em andamento. Se nenhuma exceção estiver em andamento, a função retornará um objeto `exception_ptr` que não esteja associado a nenhuma exceção.

A função `make_exception_ptr` retorna um `exception_ptr` objeto que faz referência à exceção especificada pelo parâmetro `e`.

Comentários

Cenário

Imagine que você deseja criar um aplicativo que possa ser dimensionado para lidar com uma quantidade variável de trabalho. Para atingir esse objetivo, você projeta um aplicativo multi-threaded em que um thread primário inicializa o número de threads secundários necessários para que o trabalho possa ser realizado. Os threads secundários ajudam o thread primário a gerenciar recursos, balancear as cargas e aprimorar o rendimento. Ao distribuir o

trabalho, o aplicativo multi-threaded apresenta melhor desempenho do que um aplicativo single-threaded.

No entanto, se um thread secundário lançar uma exceção, você quer que o thread primário trate dela. Isso porque você deseja que o aplicativo trate das exceções de maneira consistente e unificada, independentemente do número de threads secundários.

Solução

Para atender ao objetivo do cenário anterior, o C++ Standard oferece suporte ao transporte de uma exceção entre threads. Se um thread secundário lançar uma exceção, essa exceção se tornará a *exceção atual*. Por analogia com o mundo real, a exceção atual é considerada *em trânsito*. A exceção atual estará em voo do momento em que é lançada até o momento em que o manipulador de exceção que a captura retorno.

O thread secundário pode capturar a exceção atual em um `catch` bloco e, em seguida, chamar a `current_exception` função para armazenar a exceção em um `exception_ptr` objeto. O objeto `exception_ptr` deve estar disponível para o thread secundário e o thread primário. Por exemplo, o objeto `exception_ptr` pode ser uma variável global cujo acesso é controlado por um mutex. O termo *transporte uma exceção* significa que uma exceção em um thread pode ser convertida em um formulário que pode ser acessado por outro thread.

Em seguida, o thread primário chama a função `rethrow_exception`, que o extrai e lança a exceção do objeto `exception_ptr`. Quando a exceção é lançada, ela se torna a exceção atual no thread primário. Isto é, ela parece se originar no thread primário.

Por fim, o thread principal pode capturar a exceção atual em um `catch` bloco e, em seguida, processá-la ou emití-la para um manipulador de exceção de nível mais alto. Ou, o thread primário pode ignorar a exceção e permitir que o processo seja encerrado.

A maioria dos aplicativos não precisa transportar exceções entre os threads. No entanto, esse recurso é útil em um sistema de computação paralela, pois o sistema pode dividir o trabalho entre os threads secundários, processadores ou núcleos. Em um ambiente de computação paralela, um thread único e dedicado pode tratar todas as exceções dos threads secundários, além de poder apresentar um modelo de tratamento de exceções consistente para qualquer aplicativo.

Para obter mais informações sobre a proposta da comissão do C++ Standard, procure na Internet pelo número de documento N2179, denominado "Language Support for Transporting Exceptions between Threads".

Modelos de manipulação de exceções e opções de compilador

O modelo de tratamento de exceções do seu aplicativo determina se ele pode capturar e transportar uma exceção. O Visual C++ oferece suporte a três modelos que podem tratar de exceções do C++, exceções SEH (tratamento de exceções estruturadas) e exceções CLR (common language runtime). Use as opções do compilador `/eh` e `/CLR` para especificar o modelo de tratamento de exceção do aplicativo.

Somente as opções de combinação de compiladores e as instruções de programação a seguir podem transportar uma exceção. Outras combinações, ou não podem capturar exceções, ou podem capturá-las, mas não podem transportá-las.

- A opção de compilador `/EHA` e a `catch` instrução podem transportar Exceções SEH e C++.
- As opções do compilador `/EHA`, o `/EHSe` `/EHsc` e a `catch` instrução podem transportar exceções de C++.
- A opção de compilador `/CLR` e a `catch` instrução podem transportar exceções de C++. A opção de compilador `/CLR` implica a especificação da opção `/EHA`. Observe que o compilador não oferece suporte ao transporte de exceções gerenciadas. Isso ocorre porque as exceções gerenciadas, que são derivadas da classe `System.Exception`, já são objetos que você pode mover entre threads usando os recursos do Common language Runtime.

IMPORTANT

Recomendamos que você especifique a opção de compilador /EHsc e Capture apenas as exceções de C++. Você se expõe a uma ameaça de segurança se usar a opção de compilador /EHA ou /CLR e uma `catch` instrução com uma *declaração de exceção de reticências* (`catch(...)`). Provavelmente, você pretende usar a `catch` instrução para capturar algumas exceções específicas. No entanto, a instrução `catch(...)` captura todas as exceções C++ e SEH, incluindo as inesperadas que devem fatais. Se você ignorar ou tratar incorretamente uma exceção inesperada, o código mal-intencionado poderá aproveitar essa oportunidade para destruir a segurança do seu programa.

Uso

As seções a seguir descrevem como transportar exceções usando o `exception_ptr` tipo, e as `current_exception` funções, `rethrow_exception` e `make_exception_ptr`.

tipo de exception_ptr

Use um objeto `exception_ptr` para fazer referência à exceção atual ou a uma instância de uma exceção especificada pelo usuário. Na implementação da Microsoft, uma exceção é representada por uma estrutura `EXCEPTION_RECORD`. Cada objeto `exception_ptr` inclui um campo de referência de exceção que aponta para uma cópia da estrutura `EXCEPTION_RECORD` que representa a exceção.

Quando você declara uma variável `exception_ptr`, ela não é associada a nenhuma exceção. Isto é, o campo de referência de exceção é NULL. Esse objeto `exception_ptr` é chamado de *null exception_ptr*.

Use a função `current_exception` ou `make_exception_ptr` para atribuir uma exceção a um objeto `exception_ptr`. Quando você atribui uma exceção a uma variável `exception_ptr`, o campo de referência de exceção da variável aponta para uma cópia da exceção. Se não houver memória suficiente para copiar a exceção, o campo de referência de exceção apontará para uma cópia de uma exceção `std::bad_alloc`. Se a `current_exception` `make_exception_ptr` função ou não puder copiar a exceção por qualquer outro motivo, a função chamará a função `Terminate` para sair do processo atual.

Apesar do nome, um objeto `exception_ptr` não é, em si, um ponteiro. Ele não obedece à semântica do ponteiro e não pode ser usado com os operadores de acesso do membro do ponteiro (`->`) ou de indireção (`*`). O objeto `exception_ptr` não tem membros de dados públicos ou funções de membro.

Comparações

Você pode usar os operadores igual (`==`) e diferente (`!=`) para comparar dois objetos `exception_ptr`. Os operadores não comparam o valor binário (padrão de bit) das estruturas `EXCEPTION_RECORD` que representam as exceções. Em vez disso, os operadores comparam os endereços no campo de referência de exceção dos objetos `exception_ptr`. Consequentemente, um `exception_ptr` nulo e o valor NULL são comparados como iguais.

função current_exception

Chame a `current_exception` função em um `catch` bloco. Se uma exceção estiver em trânsito e o `catch` bloco puder capturar a exceção, a `current_exception` função retornará um `exception_ptr` objeto que faz referência à exceção. Caso contrário, a função retornará um objeto `exception_ptr` nulo.

Detalhes

A `current_exception` função captura a exceção que está em trânsito, independentemente de a `catch` instrução especificar uma instrução [de declaração de exceção](#).

O destruidor para a exceção atual será chamado no final do `catch` bloco se você não relançar a exceção. No entanto, mesmo que você chame a função `current_exception` no destruidor, a função retornará um objeto

`exception_ptr` que faz referência à exceção atual.

As chamadas sucessivas à função `current_exception` retornam objetos `exception_ptr` que se referem a diferentes cópias da exceção atual. Consequentemente, os objetos são comparados como diferentes, pois se referem a diferentes cópias, mesmo quando as cópias têm o mesmo valor binário.

Exceções SEH

Se você usar a opção de compilador `/EHA`, poderá capturar uma exceção SEH em um `catch` bloco C++. A função `current_exception` retorna um objeto `exception_ptr` que faz referência à exceção SEH. E a `rethrow_exception` função lançará a exceção SEH se você chamá-la com o objeto transportado `exception_ptr` como seu argumento.

A `current_exception` função retornará um NULL `exception_ptr` se você chamá-lo em um `__finally` manipulador de término Seh, um `__except` manipulador de exceção ou a `__except` expressão de filtro.

Uma exceção transportada não oferece suporte a exceções aninhadas. Uma exceção aninhada ocorrerá se outra exceção for lançada enquanto uma exceção estiver sendo tratada. Se você capturar uma exceção aninhada, o membro de dados `EXCEPTION_RECORD.ExceptionRecord` apontará para uma cadeia de estruturas `EXCEPTION_RECORD` que descreve as exceções associadas. A função `current_exception` não oferece suporte a exceções aninhadas, pois ela retorna um objeto `exception_ptr` cujo membro de dados `ExceptionRecord` é zerado.

Se você capturar uma exceção SEH, será preciso gerenciar a memória referenciada por qualquer ponteiro na matriz de membros de dados `EXCEPTION_RECORD.ExceptionInformation`. É preciso garantir que a memória seja válida durante o tempo de vida do objeto `exception_ptr` correspondente e que seja alimentada quando o objeto `exception_ptr` for excluído.

É possível usar funções de conversão de SE (exceção estruturada) juntamente com o recurso de transporte de exceções. Se uma exceção SEH for convertida em uma exceção C++, a função `current_exception` retornará um `exception_ptr` que fará referência à exceção convertida, e não à exceção SEH original. A função `rethrow_exception` lança subsequentemente a exceção convertida, não a exceção original. Para obter mais informações sobre as funções do tradutor SE, consulte [_set_se_translator](#).

função `rethrow_exception`

Depois de armazenar uma exceção capturada em um objeto `exception_ptr`, o thread primário poderá processar o objeto. Em seu thread primário, chame a função `rethrow_exception` juntamente com o objeto `exception_ptr` como seu argumento. A função `rethrow_exception` extraí a exceção do objeto `exception_ptr` e a lança no contexto do thread primário. Se o parâmetro `p` da `rethrow_exception` função for NULL `exception_ptr`, a função lançará [std::bad_exception](#).

A exceção extraída agora é a exceção atual no thread primário, e você pode tratá-la como faria com qualquer outra exceção. Se você capturar a exceção, poderá tratá-la imediatamente ou usar uma `throw` instrução para enviá-la a um manipulador de exceção de nível mais alto. Caso contrário, não faça nada e deixe o manipulador padrão de exceção do sistema encerrar o processo.

Função `make_exception_ptr`

A função `make_exception_ptr` usa uma instância de uma classe como seu argumento e retorna um `exception_ptr` que faz referência à instância. Normalmente, você especifica um objeto de [classe de exceção](#) como o argumento para a função `make_exception_ptr`, embora qualquer objeto de classe possa ser o argumento.

Chamar a `make_exception_ptr` função é equivalente a lançar uma exceção de C++, capturando-a em um `catch` bloco e, em seguida, chamando a `current_exception` função para retornar um `exception_ptr` objeto que faz referência à exceção. A implementação da função `make_exception_ptr` da Microsoft é mais eficiente do que lançar e depois capturar uma exceção.

Geralmente, um aplicativo não exige a função `make_exception_ptr` e não recomendamos seu uso.

Exemplo

O exemplo a seguir transporta uma exceção padrão C++ e uma exceção C++ personalizada de um thread para outro.

```
// transport_exception.cpp
// compile with: /EHsc /MD
#include <windows.h>
#include <stdio.h>
#include <exception>
#include <stdexcept>

using namespace std;

// Define thread-specific information.
#define THREADCOUNT 2
exception_ptr aException[THREADCOUNT];
int aArg[THREADCOUNT];

DWORD WINAPI ThrowExceptions( LPVOID );

// Specify a user-defined, custom exception.
// As a best practice, derive your exception
// directly or indirectly from std::exception.
class myException : public std::exception {
};

int main()
{
    HANDLE aThread[THREADCOUNT];
    DWORD ThreadID;

    // Create secondary threads.
    for( int i=0; i < THREADCOUNT; i++ )
    {
        aArg[i] = i;
        aThread[i] = CreateThread(
            NULL,           // Default security attributes.
            0,              // Default stack size.
            (LPTHREAD_START_ROUTINE) ThrowExceptions,
            (LPVOID) &aArg[i], // Thread function argument.
            0,              // Default creation flags.
            &ThreadID); // Receives thread identifier.
        if( aThread[i] == NULL )
        {
            printf("CreateThread error: %d\n", GetLastError());
            return -1;
        }
    }

    // Wait for all threads to terminate.
    WaitForMultipleObjects(THREADCOUNT, aThread, TRUE, INFINITE);
    // Close thread handles.
    for( int i=0; i < THREADCOUNT; i++ ) {
        CloseHandle(aThread[i]);
    }

    // Rethrow and catch the transported exceptions.
    for ( int i = 0; i < THREADCOUNT; i++ ) {
        try {
            if (aException[i] == NULL) {
                printf("exception_ptr %d: No exception was transported.\n", i);
            }
            else {
                rethrow_exception( aException[i] );
            }
        }
    }
}
```

```

        }
    }
    catch( const invalid_argument & ) {
        printf("exception_ptr %d: Caught an invalid_argument exception.\n", i);
    }
    catch( const myException & ) {
        printf("exception_ptr %d: Caught a myException exception.\n", i);
    }
}
}

// Each thread throws an exception depending on its thread
// function argument, and then ends.

DWORD WINAPI ThrowExceptions( LPVOID lpParam )
{
    int x = *((int*)lpParam);
    if (x == 0) {
        try {
            // Standard C++ exception.
            // This example explicitly throws invalid_argument exception.
            // In practice, your application performs an operation that
            // implicitly throws an exception.
            throw invalid_argument("A C++ exception.");
        }
        catch ( const invalid_argument & ) {
            aException[x] = current_exception();
        }
    }
    else {
        // User-defined exception.
        aException[x] = make_exception_ptr( myException() );
    }
    return TRUE;
}
}

```

```

exception_ptr 0: Caught an invalid_argument exception.
exception_ptr 1: Caught a myException exception.

```

Requisitos

Cabeçalho:<exception>

Confira também

[Tratamento de Exceção](#)

[/EH \(modelo de tratamento de exceção\)](#)

[/CLR \(compilação do Common Language Runtime\)](#)

Asserção e mensagens fornecidas pelo usuário (C++)

02/09/2020 • 3 minutes to read • [Edit Online](#)

A linguagem C++ dá suporte a três mecanismos de tratamento de erros que ajudam a depurar seu aplicativo: a diretiva `#error`, a palavra-chave `static_assert` e a macro `Assert,_assert,_wassert`. Todos os três mecanismos emitem mensagens de erro e dois também testam asserções de software. Uma asserção de software especifica uma condição que você espera ser verdadeira (true) em um ponto específico de seu programa. Se uma asserção de tempo de compilação falhar, o compilador emite uma mensagem de diagnóstico e um erro de compilação. Se uma asserção de tempo de execução falhar, o sistema operacional emite uma mensagem de diagnóstico e fecha seu aplicativo.

Comentários

O tempo de vida do seu aplicativo consiste de uma fase de pré-processamento, compilação e de tempo de execução. Cada mecanismo de tratamento de erro acessa as informações de depuração disponíveis durante uma dessas fases. Para depurar efetivamente, selecione o mecanismo que fornece as informações adequadas sobre essa fase:

- A diretiva de `#error` está em vigor no tempo de pré-processamento. Ela incondicionalmente emite uma mensagem especificada pelo usuário e causa a falha da compilação com um erro. A mensagem pode conter texto que é manipulado pelas políticas do pré-processador, mas nenhuma expressão resultante é avaliada.
- A declaração de `static_assert` está em vigor no momento da compilação. Ela testa uma asserção de software que é representada por uma expressão integral especificada pelo usuário que possa ser convertida em Booleano. Se a expressão for avaliada como zero (false), o compilador emitirá uma mensagem especificada pelo usuário e a compilação falhará com um erro.

A `static_assert` declaração é especialmente útil para depurar modelos porque os argumentos do modelo podem ser incluídos na expressão especificada pelo usuário.

- A macro `Assert,_assert,_wassert` macro está em vigor no tempo de execução. Ela avalia uma expressão especificada pelo usuário e se o resultado for zero, o sistema emitirá uma mensagem de diagnóstico e fechará seu aplicativo. Muitas outras macros, como `_ASSERT` e `_ASSERTE`, se assemelham a essa macro, mas executam mensagens de diagnóstico diferentes definidas pelo sistema ou definidas pelo usuário.

Confira também

[Diretiva de #error \(C/C++\)](#)

[Macro `Assert,_assert,_wassert`](#)

[_ASSERT, _ASSERTE, _ASSERT_EXPR macros](#)

[static_assert](#)

[_STATIC_ASSERT macro](#)

[Modelo](#)

static_assert

02/09/2020 • 5 minutes to read • [Edit Online](#)

Testa uma asserção de software no tempo de compilação. Se a expressão constante especificada for `false`, o compilador exibirá a mensagem especificada, se uma for fornecida, e a compilação falhar com o erro C2338; caso contrário, a declaração não terá nenhum efeito.

Sintaxe

```
static_assert( constant-expression, string-literal );  
static_assert( constant-expression ); // C++17 (Visual Studio 2017 and later)
```

Parâmetros

expressão de constante

Uma expressão constante integral que pode ser convertida em um valor booleano. Se a expressão avaliada for zero (false), o parâmetro *String-literal* será exibido e a compilação falhará com um erro. Se a expressão for diferente de zero (true), a `static_assert` declaração não terá nenhum efeito.

literal de cadeia de caracteres

Uma mensagem que será exibida se o parâmetro de *expressão constante* for zero. A mensagem é uma cadeia de caracteres no [conjunto de caracteres base](#) do compilador; ou seja, [caracteres não multibyte ou largos](#).

Comentários

O parâmetro *constante-expressão* de uma `static_assert` declaração representa uma *asseração de software*. Uma asserção de software especifica uma condição que você espera ser verdadeira (true) em um ponto específico de seu programa. Se a condição for verdadeira, a `static_assert` declaração não terá nenhum efeito. Se a condição for falsa, a asserção falhará, o compilador exibirá a mensagem no parâmetro *String-literal* e a compilação falhará com um erro. No Visual Studio 2017 e posterior, o parâmetro *String-literal* é opcional.

A `static_assert` declaração testa uma declaração de software no momento da compilação. Por outro lado, a [macro Assert e as funções _assert e _wassert](#) testam uma declaração de software em tempo de execução e incorrem em um custo de tempo de execução em espaço ou tempo. A `static_assert` declaração é especialmente útil para a depuração de modelos porque os argumentos de modelo podem ser incluídos no parâmetro *constante-expressão*.

O compilador examina a `static_assert` declaração de erros de sintaxe quando a declaração é encontrada. O compilador avaliará o parâmetro de *expressão constante* imediatamente se ele não depender de um parâmetro de modelo. Caso contrário, o compilador avaliará o parâmetro de *expressão constante* quando o modelo for instanciado. Consequentemente, o compilador pode emitir uma mensagem de diagnóstico uma vez quando a declaração for encontrada e novamente quando o modelo for instanciado.

Você pode usar a `static_assert` palavra-chave em namespace, classe ou escopo de bloco. (A `static_assert` palavra-chave é tecnicamente uma declaração, mesmo que não introduza um novo nome em seu programa, pois ela pode ser usada no escopo do namespace.)

Descrição

No exemplo a seguir, a `static_assert` declaração tem escopo de namespace. Como o compilador conhece o

tamanho do tipo `void *`, a expressão é avaliada imediatamente.

Exemplo

```
static_assert(sizeof(void *) == 4, "64-bit code generation is not supported.");
```

Descrição

No exemplo a seguir, a `static_assert` declaração tem escopo de classe. O `static_assert` verifica se um parâmetro de modelo é um tipo Pod (*dados antigos simples*). O compilador examina a `static_assert` declaração quando ela é declarada, mas não avalia o parâmetro de *expressão constante* até que o `basic_string` modelo de classe seja instanciado no `main()`.

Exemplo

```
#include <type_traits>
#include <iostream>
namespace std {
template <class CharT, class Traits = std::char_traits<CharT> >
class basic_string {
    static_assert(std::is_pod<CharT>::value,
                 "Template argument CharT must be a POD type in class template basic_string");
    // ...
};

struct NonPOD {
    NonPOD(const NonPOD &)
    virtual ~NonPOD() {}
};

int main()
{
    std::basic_string<char> bs;
```

Descrição

No exemplo a seguir, a `static_assert` declaração tem escopo de bloco. O `static_assert` verifica se o tamanho da estrutura `VMPage` é igual ao `pageSize` de memória virtual do sistema.

Exemplo

```
#include <sys/param.h> // defines PAGESIZE
class VMMClient {
public:
    struct VMPage { // ...
    };
    int check_pagesize() {
        static_assert(sizeof(VMPage) == PAGESIZE,
                     "Struct VMPage must be the same size as a system virtual memory page.");
        // ...
    }
    // ...
};
```

Confira também

[Asserção e mensagens fornecidas pelo usuário \(C++\)](#)

[Diretiva de #error \(C/C++\)](#)

[Macro Assert, _assert _wassert](#)

[Modelos](#)

[Conjunto de caracteres ASCII](#)

[Declarações e definições](#)

Visão geral dos módulos no C++

02/09/2020 • 14 minutes to read • [Edit Online](#)

O C++ 20 introduz *módulos*, uma solução moderna para a componentização de bibliotecas e programas C++. Um módulo é um conjunto de arquivos de código-fonte que são compilados independentemente das [unidades de tradução](#) que os importam. Os módulos eliminam ou reduzem muito muitos dos problemas associados ao uso de arquivos de cabeçalho e também reduzem os tempos de compilação. Macros, diretivas de pré-processador e nomes não exportados declarados em um módulo não são visíveis e, portanto, não têm efeito sobre a compilação da unidade de tradução que importa o módulo. Você pode importar módulos em qualquer ordem sem preocupação com redefinições de macro. As declarações na unidade de conversão de importação não participam da resolução de sobrecarga ou da pesquisa de nome no módulo importado. Depois que um módulo é compilado uma vez, os resultados são armazenados em um arquivo binário que descreve todos os tipos, funções e modelos exportados. Esse arquivo pode ser processado muito mais rápido que um arquivo de cabeçalho e pode ser reutilizado pelo compilador em todos os locais em que o módulo é importado em um projeto.

Os módulos podem ser usados lado a lado com arquivos de cabeçalho. Um arquivo de origem C++ pode importar módulos e também `#include` arquivos de cabeçalho. Em alguns casos, um arquivo de cabeçalho pode ser importado como um módulo, em vez de `#included` textualmente pelo pré-processador. Recomendamos que novos projetos usem módulos em vez de arquivos de cabeçalho o máximo possível. Para projetos maiores existentes em desenvolvimento ativo, sugerimos que você teste a conversão de cabeçalhos herdados em módulos para ver se você obtém uma redução significativa nos tempos de compilação.

Habilitar módulos no compilador do Microsoft C++

A partir do Visual Studio 2019 versão 16,2, os módulos não são totalmente implementados no compilador do Microsoft C++. Você pode usar o recurso de módulos para criar módulos de partição única e para importar os módulos de biblioteca padrão fornecidos pela Microsoft. Para habilitar o suporte para módulos, compile com `/experimental: module` e `/std: c++ Latest`. Em um projeto do Visual Studio, clique com o botão direito do mouse no nó do projeto em **Gerenciador de soluções** e escolha **Propriedades**. Defina a lista suspensa **configuração** para **todas as configurações** e escolha **Propriedades de configuração** > **linguagem C/C++ > Language > habilitar módulos C++ (experimental)**.

Um módulo e o código que o consome devem ser compilados com as mesmas opções de compilador.

Consumir a biblioteca padrão C++ como módulos

Embora não seja especificado pelo padrão C++ 20, a Microsoft permite que sua implementação da biblioteca do C++ Standard seja importada como módulos. Ao importar a biblioteca padrão C++ como módulos em vez de `#including`-la por meio de arquivos de cabeçalho, é possível acelerar os tempos de compilação dependendo do tamanho do seu projeto. A biblioteca é componentizada nos seguintes módulos:

- o STD. Regex fornece o conteúdo do cabeçalho `<regex>`
- o STD. FileSystem fornece o conteúdo do cabeçalho `<filesystem>`
- a memória padrão fornece o conteúdo do cabeçalho `<memory>`
- o padrão de Threading fornece o conteúdo dos cabeçalhos `... <atomic> <condition_variable> <future> <mutex> <shared_mutex>` e `<thread>`
- o STD. Core fornece tudo o mais na biblioteca padrão do C++

Para consumir esses módulos, basta adicionar uma declaração de importação à parte superior do arquivo de código-fonte. Por exemplo:

```
import std.core;
import std.regex;
```

Para consumir o módulo Microsoft standard library, compile seu programa com as opções [/EHsc](#) e [/MD](#).

Exemplo básico

O exemplo a seguir mostra uma definição de módulo simples em um arquivo de origem chamado `foo. IXX`. A extensão `. IXX` é necessária para arquivos de interface de módulo no Visual Studio. Neste exemplo, o arquivo de interface contém a definição de função, bem como a declaração. No entanto, as definições também podem ser colocadas em um ou mais arquivos separados (conforme mostrado em um exemplo posterior). A instrução `Export Module foo` indica que esse arquivo é a interface primária para um módulo chamado `foo`. O `export` modificador em `f()` indica que essa função estará visível quando `foo` for importada por outro programa ou módulo. Observe que o módulo faz referência a um namespace `Bar`.

```
export module Foo;

#define ANSWER 42

namespace Bar
{
    int f_internal() {
        return ANSWER;
    }

    export int f() {
        return f_internal();
    }
}
```

O arquivo `myprogram. cpp` usa a declaração de `importação` para acessar o nome que é exportado pelo `foo`. Observe que o nome `Bar` está visível aqui, mas não todos os seus membros. Observe também que a macro `ANSWER` não está visível.

```
import Foo;
import std.core;

using namespace std;

int main()
{
    cout << "The result of f() is " << Bar::f() << endl; // 42
    // int i = Bar::f_internal(); // C2039
    // int j = ANSWER; //C2065
}
```

A declaração de importação pode aparecer somente no escopo global.

Implementando módulos

Você pode criar um módulo com um único arquivo de interface (`. IXX`) que exporta nomes e inclui implementações de todas as funções e tipos. Você também pode colocar as implementações em um ou mais arquivos de implementação separados, de forma semelhante a como os arquivos `. h` e `. cpp` são usados. A `export` palavra-chave é usada somente no arquivo de interface. Um arquivo de implementação pode `importar` outro módulo, mas não pode ter `export` nenhum nome. Os arquivos de implementação podem ser nomeados com

qualquer extensão. Um arquivo de interface e o conjunto de arquivos de implementação que o fazem de volta são tratados como um tipo especial de unidade de tradução chamado *unidade de módulo*. Um nome declarado em qualquer arquivo de implementação é automaticamente visível em todos os outros arquivos dentro da mesma unidade de módulo.

Para módulos maiores, você pode dividir o módulo em várias unidades de módulo chamadas *partições*. Cada partição consiste em um arquivo de interface apoiado por um ou mais arquivos de implementação. (A partir do Visual Studio 2019 versão 16,2, as partições ainda não estão totalmente implementadas.)

Módulos, namespaces e pesquisa dependente de argumento

As regras para namespaces em módulos são as mesmas de qualquer outro código. Se uma declaração dentro de um namespace for exportada, o namespace delimitador (excluindo Membros não exportados) também será exportado implicitamente. Se um namespace for explicitamente exportado, todas as declarações nessa definição de namespace serão exportadas.

Ao executar a pesquisa dependente de argumento para resoluções de sobrecarga na unidade de conversão de importação, o compilador considera as funções declaradas na mesma unidade de tradução (incluindo as interfaces de módulo) como onde o tipo dos argumentos da função é definido.

Partições de módulo

NOTE

Esta seção é fornecida para fins de integridade. As partições ainda não foram implementadas no compilador do Microsoft C++.

Um módulo pode ser componentizado em *partições*, cada uma consistindo em um arquivo de interface e zero ou mais arquivos de implementação. Uma partição de módulo é semelhante a um módulo, exceto pelo fato de que ela compartilha a propriedade de todas as declarações em todo o módulo. Todos os nomes exportados por arquivos de interface de partição são importados e exportados novamente pelo arquivo de interface primário. O nome de uma partição deve começar com o nome do módulo seguido por dois-pontos. As declarações em qualquer uma das partições são visíveis no módulo inteiro. Nenhuma precaução especial é necessária para evitar erros de ODR (um-Definition-Rule). Você pode declarar um nome (função, classe, etc.) em uma partição e defini-la em outra. Um arquivo de implementação de partição começa da seguinte maneira:

```
module Foo:part1
```

e o arquivo da interface de partição começa desta forma:

```
export module Foo:part1
```

Para acessar declarações em outra partição, uma partição deve importá-la, mas ela só pode usar o nome da partição, não o nome do módulo:

```
module Foo:part2;
import :part1;
```

A unidade de interface primária deve importar e reexportar todos os arquivos de partição da interface do módulo da seguinte maneira:

```
export import :part1
export import :part2
...
```

A unidade de interface primária pode importar arquivos de implementação de partição, mas não pode exportá-los porque esses arquivos não têm permissão para exportar nomes. Isso permite que um módulo mantenha detalhes de implementação internos ao módulo.

Módulos e arquivos de cabeçalho

Você pode incluir arquivos de cabeçalho em um arquivo de origem de módulo colocando a `#include` diretiva antes da declaração do módulo. Esses arquivos são considerados no *fragmento do módulo global*. Um módulo só pode ver os nomes no *fragmento do módulo global* que estão nos cabeçalhos que ele inclui explicitamente. O fragmento do módulo global contém apenas símbolos que são realmente usados.

```
// MyModuleA.cpp

#include "customlib.h"
#include "anotherlib.h"

import std.core;
import MyModuleB;

//... rest of file
```

Você pode usar um arquivo de cabeçalho tradicional para controlar quais módulos são importados:

```
// MyProgram.h
import std.core;
#ifndef DEBUG_LOGGING
import std.filesystem;
#endif
```

Arquivos de cabeçalho importados

NOTE

Esta seção é apenas informativa. As importações herdadas ainda não foram implementadas no compilador do Microsoft C++.

Alguns cabeçalhos são suficientemente independentes que podem ser trazidos usando a palavra-chave **Import**. A principal diferença entre um cabeçalho importado e um módulo importado é que qualquer definição de pré-processador no cabeçalho é visível no programa de importação imediatamente após a instrução de importação. (As definições de pré-processador em quaisquer arquivos incluídos por esse cabeçalho *não* são visíveis.)

```
import <vector>
import "myheader.h"
```

Confira também

[module, import e export](#)

module, import e export

02/09/2020 • 2 minutes to read • [Edit Online](#)

O **módulo**, a **importação** e as `export` declarações estão disponíveis no C++ 20 e exigem a opção de compilador `/experimental:module` juntamente com `/std: C++ mais recente`. Para obter mais informações, consulte [visão geral dos módulos em C++](#).

module

Coloque uma declaração de **módulo** no início de um arquivo de implementação de módulo para especificar que o conteúdo do arquivo pertence ao módulo nomeado.

```
module ModuleA;
```

exportar

Use uma declaração de **módulo de exportação** para o arquivo de interface principal do módulo, que deve ter a extensão `.Ixx`:

```
export module ModuleA;
```

Em um arquivo de interface, use o `export` modificador em nomes que se destinam a fazer parte da interface pública:

```
// ModuleA.Ixx

export module ModuleA;

namespace Bar
{
    export int f();
    export double d();
    double internal_f(); // not exported
}
```

Nomes não exportados não são visíveis para o código que importa o módulo:

```
//MyProgram.cpp

import module ModuleA;

int main() {
    Bar::f(); // OK
    Bar::d(); // OK
    Bar::internal_f(); // Ill-formed: error C2065: 'internal_f': undeclared identifier
}
```

A `export` palavra-chave pode não aparecer em um arquivo de implementação de módulo. Quando `export` é aplicado a um nome de namespace, todos os nomes no namespace são exportados.

import

Use uma declaração de **importação** para tornar os nomes de um módulo visíveis em seu programa. A declaração de importação deve aparecer após a declaração de módulo e depois de qualquer `#include` diretivas, mas antes de qualquer declaração no arquivo.

```
module ModuleA;

#include "custom-lib.h"
import std.core;
import std.regex;
import ModuleB;

// begin declarations here:
template <class T>
class Baz
{...};
```

Comentários

Tanto a **importação** quanto o **módulo** são tratados como palavras-chave somente quando aparecem no início de uma linha lógica:

```
// OK:
module ;
module module-name
import :
import <
import "
import module-name
export module ;
export module module-name
export import :
export import <
export import "
export import module-name

// Error:
int i; module ;
```

Específico da Microsoft

No Microsoft C++, a **importação** e o **módulo** de tokens são sempre identificadores e nunca palavras-chave quando são usados como argumentos para uma macro.

Exemplo

```
#define foo(...) __VA_ARGS__
foo(
import // Always an identifier, never a keyword
)
```

Fim da seção específica da Microsoft

Consulte Também

[Visão geral dos módulos no C++](#)

Modelos (C++)

02/09/2020 • 12 minutes to read • [Edit Online](#)

Os modelos são a base para a programação genérica em C++. Como uma linguagem fortemente tipada, o C++ requer que todas as variáveis tenham um tipo específico, explicitamente declarado pelo programador ou deduzido pelo compilador. No entanto, muitas estruturas de dados e algoritmos têm a mesma aparência, independentemente do tipo em que estão operando. Os modelos permitem que você defina as operações de uma classe ou função e permita que o usuário especifique em quais tipos concretos essas operações devem trabalhar.

Definindo e usando modelos

Um modelo é um constructo que gera um tipo ou função comum em tempo de compilação com base nos argumentos que o usuário fornece para os parâmetros de modelo. Por exemplo, você pode definir um modelo de função como este:

```
template <typename T>
T minimum(const T& lhs, const T& rhs)
{
    return lhs < rhs ? lhs : rhs;
}
```

O código acima descreve um modelo para uma função genérica com um parâmetro de tipo único *T*, cujo valor de retorno e parâmetros de chamada (LHS e RHS) são todos desse tipo. Você pode nomear um parâmetro de tipo como desejar, mas por convenção letras maiúsculas únicas são usadas com mais frequência. *T* é um parâmetro de modelo; a `typename` palavra-chave indica que esse parâmetro é um espaço reservado para um tipo. Quando a função é chamada, o compilador substituirá cada instância do `T` pelo argumento de tipo concreto que é especificado pelo usuário ou deduzido pelo compilador. O processo no qual o compilador gera uma classe ou função de um modelo é conhecida como *instanciação de modelo*; `minimum<int>` é uma instanciação do modelo `minimum<T>`.

Em outro lugar, um usuário pode declarar uma instância do modelo especializado para `int`. suponha que `get_a()` e `get_b()` sejam funções que retornam um `int`:

```
int a = get_a();
int b = get_b();
int i = minimum<int>(a, b);
```

No entanto, como esse é um modelo de função e o compilador pode deduzir o tipo dos `T` argumentos *a* e *b*, você pode chamá-lo exatamente como uma função comum:

```
int i = minimum(a, b);
```

Quando o compilador encontra essa última instrução, ele gera uma nova função na qual cada ocorrência de *T* no modelo é substituída por `int` :

```
int minimum(const int& lhs, const int& rhs)
{
    return lhs < rhs ? lhs : rhs;
}
```

As regras de como o compilador executa a dedução de tipo nos modelos de função baseiam-se nas regras para funções comuns. Para obter mais informações, consulte [sobrecarga de chamadas de modelo de função](#).

Parâmetros de tipo

No `minimum` modelo acima, observe que o parâmetro de tipo *T* não está qualificado de nenhuma forma até que seja usado nos parâmetros de chamada de função, onde os qualificadores `const` e `Reference` são adicionados.

Não há nenhum limite prático para o número de parâmetros de tipo. Separe vários parâmetros por vírgulas:

```
template <typename T, typename U, typename V> class Foo{};
```

A palavra-chave `class` é equivalente a `typename` neste contexto. Você pode expressar o exemplo anterior como:

```
template <class T, class U, class V> class Foo{};
```

Você pode usar o operador de reticências (...) para definir um modelo que usa um número arbitrário de zero ou mais parâmetros de tipo:

```
template<typename... Arguments> class vtclass;

vtclass< > vtinstance1;
vtclass<int> vtinstance2;
vtclass<float, bool> vtinstance3;
```

Qualquer tipo interno ou definido pelo usuário pode ser usado como um argumento de tipo. Por exemplo, você pode usar `std::vector` na biblioteca padrão para armazenar variáveis do tipo `int`, `double`, `std::String`, `MyClass`, `const MyClass *`, `MyClass&` e assim por diante. A restrição principal ao usar modelos é que um argumento de tipo deve dar suporte a quaisquer operações que são aplicadas aos parâmetros de tipo. Por exemplo, se chamarmos `minimum` usando `MyClass` as neste exemplo:

```
class MyClass
{
public:
    int num;
    std::wstring description;
};

int main()
{
    MyClass mc1 {1, L"hello"};
    MyClass mc2 {2, L"goodbye"};
    auto result = minimum(mc1, mc2); // Error! C2678
}
```

Um erro de compilador será gerado porque `MyClass` o não fornece uma sobrecarga para o `<` operador.

Não há nenhum requisito inerente de que os argumentos de tipo para qualquer modelo específico pertençam à mesma hierarquia de objetos, embora você possa definir um modelo que impõe tal restrição. Você pode

combinar técnicas orientadas a objeto com modelos; por exemplo, você pode armazenar um `*` derivado em um vetor `<Base*>`. Observe que os argumentos devem ser ponteiros

```
vector<MyClass*> vec;
MyDerived d(3, L"back again", time(0));
vec.push_back(&d);

// or more realistically:
vector<shared_ptr<MyClass>> vec2;
vec2.push_back(make_shared<MyDerived>());
```

Os requisitos básicos que `std::vector` e outros contêineres de biblioteca padrão impõem em elementos `T` é que são `T` Copy-atribuível e Copy-constructível.

Parâmetros sem tipo

Ao contrário dos tipos genéricos em outras linguagens, como C# e Java, os modelos do C++ dão suporte a *parâmetros sem tipo*, também chamados de parâmetros de valor. Por exemplo, você pode fornecer um valor integral constante para especificar o comprimento de uma matriz, como com este exemplo semelhante à classe `std::array` na biblioteca padrão:

```
template<typename T, size_t L>
class MyArray
{
    T arr[L];
public:
    MyArray() { ... }
};
```

Observe a sintaxe na declaração de modelo. O `size_t` valor é passado como um argumento de modelo no momento da compilação e deve ser `const` ou uma `constexpr` expressão. Você o usa da seguinte maneira:

```
MyArray<MyClass*, 10> arr;
```

Outros tipos de valores, incluindo ponteiros e referências, podem ser passados como parâmetros sem tipo. Por exemplo, você pode passar um ponteiro para um objeto `Function` ou `Function` para personalizar alguma operação dentro do código do modelo.

Dedução de tipo para parâmetros de modelo sem tipo

No Visual Studio 2017 e posterior, em `/std: modo c++ 17` o compilador deduz o tipo de um argumento de modelo não tipo declarado com `auto`:

```
template <auto x> constexpr auto constant = x;

auto v1 = constant<5>;      // v1 == 5, decltype(v1) is int
auto v2 = constant<true>;    // v2 == true, decltype(v2) is bool
auto v3 = constant<'a'>;     // v3 == 'a', decltype(v3) is char
```

Modelos como parâmetros de modelo

Um modelo pode ser um parâmetro de modelo. Neste exemplo, `MyClass2` tem dois parâmetros de template: um `typeName` de parâmetro `T` e um parâmetro de template `arr`.

```
template<typename T, template<typename U, int I> class Arr>
class MyClass2
{
    T t; //OK
    Arr<T, 10> a;
    U u; //Error. U not in scope
};
```

Como o próprio parâmetro *arr* não tem corpo, seus nomes de parâmetro não são necessários. Na verdade, é um erro fazer referência aos nomes de parâmetro de classe ou TypeName de *arr* dentro do corpo de `MyClass2`. Por esse motivo, os nomes de parâmetro de tipo de *arr* podem ser omitidos, conforme mostrado neste exemplo:

```
template<typename T, template<typename, int> class Arr>
class MyClass2
{
    T t; //OK
    Arr<T, 10> a;
};
```

Argumentos de modelo padrão

Os modelos de classe e função podem ter argumentos padrão. Quando um modelo tem um argumento padrão, você pode deixá-lo não especificado ao usá-lo. Por exemplo, o modelo `std::vector` tem um argumento padrão para o alocador:

```
template <class T, class Allocator = allocator<T>> class vector;
```

Na maioria dos casos, a classe `std::allocator` é aceitável e, portanto, você usa um vetor como este:

```
vector<int> myInts;
```

Mas, se necessário, você pode especificar um alocador personalizado como este:

```
vector<int, MyAllocator> ints;
```

Para mais argumentos de modelo, todos os argumentos após o primeiro argumento padrão devem ter argumentos padrão.

Ao usar um modelo cujos parâmetros estão todos padronizados, use colchetes de ângulo vazios:

```
template<typename A = int, typename B = double>
class Bar
{
    //...
};

int main()
{
    Bar<> bar; // use all default type arguments
}
```

Especialização de modelo

Em alguns casos, não é possível ou desejável que um modelo defina exatamente o mesmo código para qualquer tipo. Por exemplo, você pode desejar definir um caminho de código a ser executado somente se o argumento de tipo for um ponteiro ou um std:: wstring ou um tipo derivado de uma classe base específica. Nesses casos, você pode definir uma *especialização* do modelo para esse tipo específico. Quando um usuário instancia o modelo com esse tipo, o compilador usa a especialização para gerar a classe e, para todos os outros tipos, o compilador escolhe o modelo mais geral. As especializações nas quais todos os parâmetros são especializados são *especializações completas*. Se apenas alguns dos parâmetros são especializados, ele é chamado de *especialização parcial*.

```
template <typename K, typename V>
class MyMap{/*...*/};

// partial specialization for string keys
template<typename V>
class MyMap<string, V> {/*...*/};
...
MyMap<int, MyClass> classes; // uses original template
MyMap<string, MyClass> classes2; // uses the partial specialization
```

Um modelo pode ter qualquer número de especializações, desde que cada parâmetro de tipo especializado seja exclusivo. Somente modelos de classe podem ser parcialmente especializados. Todas as especializações completas e parciais de um modelo devem ser declaradas no mesmo namespace que o modelo original.

Para obter mais informações, consulte [especialização de modelo](#).

typename

02/09/2020 • 2 minutes to read • [Edit Online](#)

Em definições de modelo, fornece uma dica ao compilador de que um identificador desconhecido é um tipo. Em listas de parâmetros de modelo, é usado para especificar um parâmetro de tipo.

Sintaxe

```
typename identifier;
```

Comentários

Essa palavra-chave deve ser usada se um nome em uma definição de modelo for um nome qualificado que dependa de um argumento de modelo; é opcional se o nome qualificado não for dependente. Para obter mais informações, consulte [modelos e resolução de nomes](#).

`typename` pode ser usado por qualquer tipo em qualquer lugar em uma definição ou declaração de modelo. Não é permitido na lista de classes base, a menos que como um argumento de modelo para uma classe base de modelo.

```
template <class T>
class C1 : typename T::InnerType // Error - typename not allowed.
{};
template <class T>
class C2 : A<typename T::InnerType> // typename OK.
{};


```

A `typename` palavra-chave também pode ser usada no lugar de `class` listas de parâmetros de modelo. Por exemplo, as instruções a seguir são semanticamente equivalentes:

```
template<class T1, class T2>...
template<typename T1, typename T2>...
```

Exemplo

```
// typename.cpp
template<class T> class X
{
    typename T::Y m_y;    // treat Y as a type
};

int main()
{
}
```

Confira também

[Modelo](#)

[Palavras-chave](#)

Modelos de classe

25/03/2020 • 9 minutes to read • [Edit Online](#)

Este tópico descreve as regras específicas para modelos C++ de classe.

Funções de membro de modelos de classe

As funções de membro podem ser definidas dentro ou fora de um modelo de classe. Elas são definidas como modelos de função se definidas fora do modelo de classe.

```
// member_function_templates1.cpp
template<class T, int i> class MyStack
{
    T* pStack;
    T StackBuffer[i];
    static const int cItems = i * sizeof(T);
public:
    MyStack( void );
    void push( const T item );
    T& pop( void );
};

template< class T, int i > MyStack< T, i >::MyStack( void )
{
};

template< class T, int i > void MyStack< T, i >::push( const T item )
{
};

template< class T, int i > T& MyStack< T, i >::pop( void )
{
};

int main()
{
}
```

Observe que, como em qualquer função de membro da classe de modelo, a definição da função de membro do construtor da classe inclui a lista de argumentos de modelo duas vezes.

As próprias funções de membro podem ser modelos de função, especificando parâmetros adicionais, como no exemplo a seguir.

```

// member_templates.cpp
template<typename T>
class X
{
public:
    template<typename U>
    void mf(const U &u);
};

template<typename T> template <typename U>
void X<T>::mf(const U &u)
{
}

int main()
{
}

```

Modelos de classe aninhada

Os modelos podem ser definidos dentro de classes ou de modelos de classe. Nesse caso, eles são chamados de modelos de membro. Os modelos de membro que são classes são chamados de modelos de classe aninhados. Os modelos de membro que são funções são abordados em [modelos de função de membro](#).

Os modelos de classe aninhados são declarados como modelos de classe dentro do escopo da classe externa. Eles podem ser definidos dentro ou fora da classe delimitadora.

O código a seguir demonstra um modelo de classe aninhado em uma classe comum.

```

// nested_class_template1.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

class X
{
    template <class T>
    struct Y
    {
        T m_t;
        Y(T t): m_t(t) { }
    };

    Y<int> yInt;
    Y<char> yChar;

public:
    X(int i, char c) : yInt(i), yChar(c) { }
    void print()
    {
        cout << yInt.m_t << " " << yChar.m_t << endl;
    }
};

int main()
{
    X x(1, 'a');
    x.print();
}

```

```
// nested_class_template2.cpp
```

```

// compile with: /EHsc
#include <iostream>
using namespace std;

template <class T>
class X
{
    template <class U> class Y
    {
        U* u;
    public:
        Y();
        U& Value();
        void print();
        ~Y();
    };
    Y<int> y;
public:
    X(T t) { y.Value() = t; }
    void print() { y.print(); }
};

template <class T>
template <class U>
X<T>::Y<U>::Y()
{
    cout << "X<T>::Y<U>::Y()" << endl;
    u = new U();
}

template <class T>
template <class U>
U& X<T>::Y<U>::Value()
{
    return *u;
}

template <class T>
template <class U>
void X<T>::Y<U>::print()
{
    cout << this->Value() << endl;
}

template <class T>
template <class U>
X<T>::Y<U>::~Y()
{
    cout << "X<T>::Y<U>::~Y()" << endl;
    delete u;
}

int main()
{
    X<int>* xi = new X<int>(10);
    X<char>* xc = new X<char>('c');
    xi->print();
    xc->print();
    delete xi;
    delete xc;
}

//Output:
X<T>::Y<U>::Y()
X<T>::Y<U>::Y()
10
99
X<T>::Y<U>::~Y()

```

As classes locais não podem ter modelos de membro.

Amigos do modelo

Os modelos de classe podem ter [amigos](#). Uma classe ou um modelo de classe, uma função ou um modelo de função podem ser amigas (friends) de uma classe de modelo. Friends também podem ser especializações de um modelo de classe ou modelo de função, mas não especializações parciais.

No exemplo a seguir, uma função friend é definida como um modelo de função no modelo de classe. Esse código gera uma versão da função friend para cada instanciação do modelo. Essa construção é útil quando a função friend depende dos mesmos parâmetros de modelo que a classe.

```
// template_friend1.cpp
// compile with: /EHsc

#include <iostream>
using namespace std;

template <class T> class Array {
    T* array;
    int size;

public:
    Array(int sz): size(sz) {
        array = new T[size];
        memset(array, 0, size * sizeof(T));
    }

    Array(const Array& a) {
        size = a.size;
        array = new T[size];
        memcpy_s(array, a.array, sizeof(T));
    }

    T& operator[](int i) {
        return *(array + i);
    }

    int Length() { return size; }

    void print() {
        for (int i = 0; i < size; i++)
            cout << *(array + i) << " ";

        cout << endl;
    }
};

template<class T>
friend Array<T>* combine(Array<T>& a1, Array<T>& a2);
};

template<class T>
Array<T>* combine(Array<T>& a1, Array<T>& a2) {
    Array<T>* a = new Array<T>(a1.size + a2.size);
    for (int i = 0; i < a1.size; i++)
        (*a)[i] = *(a1.array + i);

    for (int i = 0; i < a2.size; i++)
        (*a)[i + a1.size] = *(a2.array + i);

    return a;
}
```

```

int main() {
    Array<char> alpha1(26);
    for (int i = 0 ; i < alpha1.Length() ; i++)
        alpha1[i] = 'A' + i;

    alpha1.print();

    Array<char> alpha2(26);
    for (int i = 0 ; i < alpha2.Length() ; i++)
        alpha2[i] = 'a' + i;

    alpha2.print();
    Array<char>*alpha3 = combine(alpha1, alpha2);
    alpha3->print();
    delete alpha3;
}

//Output:
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z a b c d e f g h i j k l m n o p q r s t u v w x y z

```

O exemplo a seguir envolve uma função friend que tem uma especialização de modelo. Uma especialização de modelo de função é automaticamente friend quando o modelo de função original é friend.

Também é possível declarar apenas a versão especializada do modelo como o friend, como o comentário antes da declaração friend no código a seguir indica. Se você fizer isso, deverá colocar a definição de especialização do modelo friend fora da classe de modelo.

```

// template_friend2.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

template <class T>
class Array;

template <class T>
void f(Array<T>& a);

template <class T> class Array
{
    T* array;
    int size;

public:
    Array(int sz): size(sz)
    {
        array = new T[size];
        memset(array, 0, size * sizeof(T));
    }
    Array(const Array& a)
    {
        size = a.size;
        array = new T[size];
        memcpy_s(array, a.array, sizeof(T));
    }
    T& operator[](int i)
    {
        return *(array + i);
    }
    int Length()
    {
        return size;
    }
    void print()
    {

```

```

        for (int i = 0; i < size; i++)
        {
            cout << *(array + i) << " ";
        }
        cout << endl;
    }

    // If you replace the friend declaration with the int-specific
    // version, only the int specialization will be a friend.
    // The code in the generic f will fail
    // with C2248: 'Array<T>::size' :
    // cannot access private member declared in class 'Array<T>'.
    //friend void f<int>(Array<int>& a);

    friend void f<>(Array<T>& a);
};

// f function template, friend of Array<T>
template <class T>
void f(Array<T>& a)
{
    cout << a.size << " generic" << endl;
}

// Specialization of f for int arrays
// will be a friend because the template f is a friend.
template<> void f(Array<int>& a)
{
    cout << a.size << " int" << endl;
}

int main()
{
    Array<char> ac(10);
    f(ac);

    Array<int> a(10);
    f(a);
}
//Output:
10 generic
10 int

```

O exemplo a seguir mostra um modelo de classe friend declarado em um modelo de classe. O modelo de classe é usado como o argumento de modelo para a classe friend. Os modelos de classe friend devem ser definidos fora do modelo de classe em que são declarados. Todas as especializações ou as especializações parciais do modelo friend também são amigas (friends) do modelo de classe original.

```

// template_friend3.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

template <class T>
class X
{
private:
    T* data;
    void InitData(int seed) { data = new T(seed); }
public:
    void print() { cout << *data << endl; }
    template <class U> friend class Factory;
};

template <class U>
class Factory
{
public:
    U* GetNewObject(int seed)
    {
        U* pu = new U;
        pu->InitData(seed);
        return pu;
    }
};

int main()
{
    Factory< X<int> > XintFactory;
    X<int>* x1 = XintFactory.GetNewObject(65);
    X<int>* x2 = XintFactory.GetNewObject(97);

    Factory< X<char> > XcharFactory;
    X<char>* x3 = XcharFactory.GetNewObject(65);
    X<char>* x4 = XcharFactory.GetNewObject(97);
    x1->print();
    x2->print();
    x3->print();
    x4->print();
}
//Output:
65
97
A
a

```

Reutilização de parâmetros de modelo

Os parâmetros de modelo podem ser reutilizados na lista de parâmetros de modelo. Por exemplo, o código a seguir é permitido:

```
// template_specifications2.cpp

class Y
{
};

template<class T, T* pT> class X1
{
};

template<class T1, class T2 = T1> class X2
{
};

Y aY;

X1<Y, &aY> x1;
X2<int> x2;

int main()
{
}
```

Confira também

[Modelos](#)

Modelos de função

02/09/2020 • 3 minutes to read • [Edit Online](#)

Os modelos de classe definem uma família de classes relacionadas que se baseiam nos argumentos de tipo passados para a classe na instanciação. Os modelos de função são semelhantes aos modelos de classe, mas definem uma família de funções. Com modelos de função, você pode especificar um conjunto de funções que se baseiam no mesmo código, mas atuam em tipos ou classes diferentes. O modelo de função a seguir permuta dois itens:

```
// function_templates1.cpp
template< class T > void MySwap( T& a, T& b ) {
    T c(a);
    a = b;
    b = c;
}
int main() {
```

Esse código define uma família de funções que permutam os valores dos argumentos. A partir desse modelo, você pode gerar funções que serão trocas `int` e `long` tipos e também tipos definidos pelo usuário. `MySwap` permutará até mesmo classes se o operador de atribuição e o construtor de cópia da classe estiverem definidos corretamente.

Além disso, o modelo de função impedirá que você permutasse objetos de tipos diferentes, porque o compilador *sabe os tipos* dos parâmetros `a` e `b` no momento da compilação.

Embora essa função possa ser executada por uma função sem modelo, usando ponteiros nulos, a versão com modelo oferece segurança de tipos. Considere as seguintes chamadas:

```
int j = 10;
int k = 18;
CString Hello = "Hello, Windows!";
MySwap( j, k );           //OK
MySwap( j, Hello );      //error
```

A segunda chamada de `MySwap` dispara um erro em tempo de compilação, pois o compilador não pode gerar uma função `MySwap` com parâmetros de tipos diferentes. Se fossem usados ponteiros nulos, as duas chamadas de função seriam compiladas corretamente, mas a função não funcionaria adequadamente em tempo de execução.

A especificação explícita dos argumentos de um modelo de função é permitida. Por exemplo:

```
// function_templates2.cpp
template<class T> void f(T) {}
int main(int j) {
    f<char>(j);    // Generate the specialization f(char).
    // If not explicitly specified, f(int) would be deduced.
}
```

Quando o argumento de modelo é especificado explicitamente, as conversões implícitas normais são realizadas para converter o argumento da função no tipo dos parâmetros correspondentes do modelo de função. No exemplo acima, o compilador converterá `j` em Type `char`.

Confira também

[Modelo](#)

[Instanciação de modelo de função](#)

[Instanciação explícita](#)

[Especialização explícita dos modelos de função](#)

Instanciação do modelo de função

25/03/2020 • 2 minutes to read • [Edit Online](#)

Quando um modelo da função é chamado primeiramente para cada tipo, o compilador cria uma instanciação. Cada instanciação é uma versão da função em modelo, especializada para o tipo. Essa instanciação será chamada sempre que a função for usada para o tipo. Se você tiver várias instanciações idênticas, mesmo em módulos diferentes, apenas uma cópia de instanciação terminará acima no arquivo executável.

A conversão de argumentos da função é permitida em modelos de função para qualquer par de argumento e de parâmetro, nos quais o parâmetro não depende de um argumento de modelo.

Os modelos da função podem ser instanciados explicitamente declarando o modelo com um tipo específico como um argumento. Por exemplo, o código a seguir é permitido:

```
// function_template_instantiation.cpp
template<class T> void f(T) { }

// Instantiate f with the explicitly specified template.
// argument 'int'
//
template void f<int> (int);

// Instantiate f with the deduced template argument 'char'.
template void f(char);
int main()
{
```

Confira também

[Modelos de função](#)

Instanciação explícita

02/09/2020 • 2 minutes to read • [Edit Online](#)

Você pode usar uma criação de instanciação explícita para criar uma instanciação de uma classe ou função com modelo sem realmente usá-la no código. Como isso é útil quando você está criando arquivos de biblioteca (.lib) que usam modelos de distribuição, definições de modelo sem instanciação não são colocadas em arquivos de objeto (.obj).

Esse código instancia explicitamente `MyStack` para `int` variáveis e seis itens:

```
template class MyStack<int, 6>;
```

Essa instrução cria uma instanciação de `MyStack` sem reservar nenhum armazenamento para um objeto. O código é gerado para todos os membros.

A linha a seguir cria instanciações explicitamente somente para a função membro do construtor:

```
template MyStack<int, 6>::MyStack( void );
```

Você pode instanciar explicitamente os modelos de função usando um argumento de tipo específico para declará-los novamente, conforme mostrado no exemplo na [Instanciação do modelo de função](#).

Você pode usar a `extern` palavra-chave para impedir a instanciação automática de membros. Por exemplo:

```
extern template class MyStack<int, 6>;
```

De forma semelhante, você pode marcar membros específicos como sendo externos e sem instanciação:

```
extern template MyStack<int, 6>::MyStack( void );
```

Você pode usar a `extern` palavra-chave para impedir que o compilador gere o mesmo código de instanciação em mais de um módulo de objeto. Você deve instanciar a função do modelo usando os parâmetros de modelo explícitos especificados em pelo menos um módulo vinculado, caso a função seja chamada, ou receberá um erro do vinculador quando o programa for criado.

NOTE

A `extern` palavra-chave na especialização se aplica somente a funções de membro definidas fora do corpo da classe. As funções definidas na declaração de classe são consideradas funções embutidas e sempre têm instanciações.

Confira também

[Modelos de função](#)

Especialização explícita de modelos de função

02/09/2020 • 2 minutes to read • [Edit Online](#)

Com um modelo de função, você pode definir o comportamento especial para um tipo específico fornecendo uma especialização explícita (substituição) do modelo da função para esse tipo. Por exemplo:

```
template<> void MySwap(double a, double b);
```

Essa declaração permite que você defina uma função diferente para `double` variáveis. Como funções que não são de modelo, as conversões de tipo padrão (como promover uma variável do tipo `float` para `double`) são aplicadas.

Exemplo

```
// explicit_specialization.cpp
template<class T> void f(T t)
{
};

// Explicit specialization of f with 'char' with the
// template argument explicitly specified:
//
template<> void f<char>(char c)
{
};

// Explicit specialization of f with 'double' with the
// template argument deduced:
//
template<> void f(double d)
{
};

int main()
{}
```

Confira também

[Modelos de função](#)

Ordenação parcial de modelos de função (C++)

02/12/2019 • 4 minutes to read • [Edit Online](#)

Vários modelos de função que correspondem à lista de argumentos de uma chamada de função podem estar disponíveis. O C++ define a ordenação parcial dos modelos de função para especificar que função deve ser chamada. A ordenação é parcial, pois pode haver alguns modelos que são considerados igualmente especializados.

O compilador escolhe a função de modelo mais especializada disponível nas correspondências possíveis. Por exemplo, se um modelo de função usa um `T` tipo e outro modelo de função `T*` que usa está disponível `T*`, a versão é considerada mais especializada. Ele é preferencial sobre a versão `T` genérica sempre que o argumento é um tipo de ponteiro, mesmo que ambos possam ser correspondências permitidas.

Use o seguinte processo para determinar se um candidato a modelo de função é mais especializado:

1. Considere dois modelos de função, T1 e T2.
2. Substitua os parâmetros em T1 por um tipo hipotético exclusivo X.
3. Com a lista de parâmetros em T1, veja se T2 é um modelo válido para essa lista de parâmetros. Ignore as conversões implícitas.
4. Repita o mesmo processo com T1 e T2 invertidos.
5. Se um modelo for uma lista de argumentos de modelo válida para o outro modelo, mas o converso não for verdadeiro, esse modelo será considerado menos especializado do que o outro modelo. Se, usando a etapa anterior, os dois modelos formarem argumentos válidos para um ao outro, eles serão considerados igualmente especializados e uma chamada ambígua resultará quando você tentar usá-los.
6. Usando estas regras:
 - a. Uma especialização de modelo para um tipo específico é mais especializada do que a que usa um argumento de tipo genérico.
 - b. Um modelo que assume `T*` apenas é mais especializado que apenas `T` um, pois um tipo `X*` hipotético é um argumento válido para um `T` argumento de modelo, `X` mas não é um argumento válido para um `T*` argumento de modelo.
 - c. `const T` é mais especializado do `T` que, `const X` porque é um argumento válido para `T` um argumento de modelo `X`, mas não é um argumento válido `const T` para um argumento de modelo.
 - d. `const T*` é mais especializado do `T*` que, `const X*` porque é um argumento válido para `T*` um argumento de modelo `X*`, mas não é um argumento válido `const T*` para um argumento de modelo.

Exemplo

O exemplo a seguir funciona conforme especificado no padrão:

```
// partial_ordering_of_function_templates.cpp
// compile with: /EHsc
#include <iostream>

template <class T> void f(T) {
    printf_s("Less specialized function called\n");
}

template <class T> void f(T*) {
    printf_s("More specialized function called\n");
}

template <class T> void f(const T*) {
    printf_s("Even more specialized function for const T*\n");
}

int main() {
    int i = 0;
    const int j = 0;
    int *pi = &i;
    const int *cpi = &j;

    f(i);    // Calls less specialized function.
    f(pi);   // Calls more specialized function.
    f(cpi); // Calls even more specialized function.
    // Without partial ordering, these calls would be ambiguous.
}
```

Saída

```
Less specialized function called
More specialized function called
Even more specialized function for const T*
```

Consulte também

[Modelos de função](#)

Modelos de função de membro

25/03/2020 • 2 minutes to read • [Edit Online](#)

O modelo de membro do termo se refere aos modelos da função de membro e modelos de classe aninhada. Os modelos da função de membro são funções de modelo que são membros de uma classe ou de um modelo de classe.

As funções de membro podem ser modelos de função em vários contextos. Todas as funções de modelos da classe são genéricas, mas não são referidas como modelos de membros ou modelos da função de membro. Se essas funções de membro usam seus próprios argumentos de modelo, são consideradas modelos da função de membro.

Exemplo

Os modelos da função de membro de classes nontemplate ou de modelo são declarados como modelos de função com seus parâmetros de modelo.

```
// member_function_templates.cpp
struct X
{
    template <class T> void mf(T* t) {}
};

int main()
{
    int i;
    X* x = new X();
    x->mf(&i);
}
```

Exemplo

O exemplo a seguir mostra um modelo da função de membro de uma classe de modelo.

```
// member_function_templates2.cpp
template<typename T>
class X
{
public:
    template<typename U>
    void mf(const U &u)
    {
    }
};

int main()
{}
```

Exemplo

```
// defining_member_templates_outside_class.cpp
template<typename T>
class X
{
public:
    template<typename U>
    void mf(const U &u);
};

template<typename T> template <typename U>
void X<T>::mf(const U &u)
{
}

int main()
{
}
```

Exemplo

As classes locais não podem ter modelos de membro.

As funções de modelo do membro não podem ser funções virtuais e não podem substituir funções virtuais de uma classe base quando são declaradas com o mesmo nome que uma função virtual da classe base.

O exemplo a seguir mostra uma conversão de modelo definida pelo usuário:

```
// templated_user_defined_conversions.cpp
template <class T>
struct S
{
    template <class U> operator S<U>()
    {
        return S<U>();
    }
};

int main()
{
    S<int> s1;
    S<long> s2 = s1; // Convert s1 using UDC and copy constructs S<long>.
}
```

Confira também

[Modelos de função](#)

Especialização de modelo (C++)

02/09/2020 • 6 minutes to read • [Edit Online](#)

Os modelos da classe podem estar parcialmente especializados e a classe resultante ainda é um modelo. A especialização parcial permite que o código do modelo seja personalizado parcialmente para tipos específicos em situações, como:

- Um modelo tem vários tipos e apenas alguns deles precisam ser especializados. O resultado é um modelo com parâmetros nos tipos restantes.
- Um modelo tem apenas um tipo, mas uma especialização é necessária para os tipos ponteiro, referência, ponteiro para o membro ou o ponteiro de função. A própria especialização ainda é um modelo no tipo apontado ou referenciado.

Exemplo

```
// partial_specialization_of_class_templates.cpp
template <class T> struct PTS {
    enum {
        IsPointer = 0,
        IsPointerToDataMember = 0
    };
};

template <class T> struct PTS<T*> {
    enum {
        IsPointer = 1,
        IsPointerToDataMember = 0
    };
};

template <class T, class U> struct PTS<T U::*> {
    enum {
        IsPointer = 0,
        IsPointerToDataMember = 1
    };
};

struct S{};

extern "C" int printf_s(const char*,...);

int main() {
    printf_s("PTS<S>::IsPointer == %d PTS<S>::IsPointerToDataMember == %d\n",
            PTS<S>::IsPointer, PTS<S>:: IsPointerToDataMember);
    printf_s("PTS<S*>::IsPointer == %d PTS<S*>::IsPointerToDataMember ==%d\n"
            , PTS<S*>::IsPointer, PTS<S*>:: IsPointerToDataMember);
    printf_s("PTS<int S::*>::IsPointer == %d PTS<int S::*>::IsPointerToDataMember == %d\n",
            PTS<int S::*>::IsPointer, PTS<int S::*>:: IsPointerToDataMember);
}
```

```
PTS<S>::IsPointer == 0 PTS<S>::IsPointerToDataMember == 0
PTS<S*>::IsPointer == 1 PTS<S*>::IsPointerToDataMember ==0
PTS<int S::*>::IsPointer == 0 PTS<int S::*>::IsPointerToDataMember == 1
```

Exemplo

Se você tiver uma classe de coleção de modelos que usa qualquer tipo `T`, poderá criar uma especialização parcial que usa qualquer tipo de ponteiro `T*`. O código a seguir demonstra um modelo de classe de coleção `Bag` e uma especialização parcial para os tipos de ponteiro nos quais a coleção diferencia os tipos de ponteiro antes de copiá-los na matriz. Então, a coleção armazena os valores que forem apontados. Com o modelo original, somente os próprios ponteiros seriam armazenados na coleção, deixando os dados vulneráveis a exclusão ou a alteração. Nesta versão especial do ponteiro da coleção, o código para verificar se há um ponteiro nulo no método `add` é adicionado.

```
// partial_specialization_of_class_templates2.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

// Original template collection class.
template <class T> class Bag {
    T* elem;
    int size;
    int max_size;

public:
    Bag() : elem(0), size(0), max_size(1) {}
    void add(T t) {
        T* tmp;
        if (size + 1 >= max_size) {
            max_size *= 2;
            tmp = new T [max_size];
            for (int i = 0; i < size; i++)
                tmp[i] = elem[i];
            tmp[size++] = t;
            delete[] elem;
            elem = tmp;
        }
        else
            elem[size++] = t;
    }

    void print() {
        for (int i = 0; i < size; i++)
            cout << elem[i] << " ";
        cout << endl;
    }
};

// Template partial specialization for pointer types.
// The collection has been modified to check for NULL
// and store types pointed to.
template <class T> class Bag<T*> {
    T* elem;
    int size;
    int max_size;

public:
    Bag() : elem(0), size(0), max_size(1) {}
    void add(T* t) {
        T* tmp;
        if (t == NULL) { // Check for NULL
            cout << "Null pointer!" << endl;
            return;
        }

        if (size + 1 >= max_size) {
            max_size *= 2;
            tmp = new T [max_size];
        }
    }
}
```

```

        for (int i = 0; i < size; i++)
            tmp[i] = elem[i];
        tmp[size++] = *t; // Dereference
        delete[] elem;
        elem = tmp;
    }
    else
        elem[size++] = *t; // Dereference
}

void print() {
    for (int i = 0; i < size; i++)
        cout << elem[i] << " ";
    cout << endl;
}
};

int main() {
    Bag<int> xi;
    Bag<char> xc;
    Bag<int*> xp; // Uses partial specialization for pointer types.

    xi.add(10);
    xi.add(9);
    xi.add(8);
    xi.print();

    xc.add('a');
    xc.add('b');
    xc.add('c');
    xc.print();

    int i = 3, j = 87, *p = new int[2];
    *p = 8;
    *(p + 1) = 100;
    xp.add(&i);
    xp.add(&j);
    xp.add(p);
    xp.add(p + 1);
    p = NULL;
    xp.add(p);
    xp.print();
}
}

```

```

10 9 8
a b c
Null pointer!
3 87 8 100

```

Exemplo

O exemplo a seguir define uma classe de modelo que usa pares de dois tipos e, em seguida, define uma especialização parcial dessa classe de modelo especializada para que um dos tipos seja `int`. A especialização define um método de classificação adicionais que implementa um tipo simples da bolha com base em inteiro.

```

// partial_specialization_of_class_templates3.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

template <class Key, class Value> class Dictionary {
    Key* keys;
    Value* values;
    int size;

```

```

int max_size;
public:
Dictionary(int initial_size) : size(0) {
    max_size = 1;
    while (initial_size >= max_size)
        max_size *= 2;
    keys = new Key[max_size];
    values = new Value[max_size];
}
void add(Key key, Value value) {
    Key* tmpKey;
    Value* tmpVal;
    if (size + 1 >= max_size) {
        max_size *= 2;
        tmpKey = new Key [max_size];
        tmpVal = new Value [max_size];
        for (int i = 0; i < size; i++) {
            tmpKey[i] = keys[i];
            tmpVal[i] = values[i];
        }
        tmpKey[size] = key;
        tmpVal[size] = value;
        delete[] keys;
        delete[] values;
        keys = tmpKey;
        values = tmpVal;
    }
    else {
        keys[size] = key;
        values[size] = value;
    }
    size++;
}

void print() {
    for (int i = 0; i < size; i++)
        cout << "{" << keys[i] << ", " << values[i] << "}" << endl;
}
};

// Template partial specialization: Key is specified to be int.
template <class Value> class Dictionary<int, Value> {
int* keys;
Value* values;
int size;
int max_size;
public:
Dictionary(int initial_size) : size(0) {
    max_size = 1;
    while (initial_size >= max_size)
        max_size *= 2;
    keys = new int[max_size];
    values = new Value[max_size];
}
void add(int key, Value value) {
    int* tmpKey;
    Value* tmpVal;
    if (size + 1 >= max_size) {
        max_size *= 2;
        tmpKey = new int [max_size];
        tmpVal = new Value [max_size];
        for (int i = 0; i < size; i++) {
            tmpKey[i] = keys[i];
            tmpVal[i] = values[i];
        }
        tmpKey[size] = key;
        tmpVal[size] = value;
        delete[] keys;
        delete[] values;
    }
}

```

```

        }
        keys = tmpKey;
        values = tmpVal;
    }
    else {
        keys[size] = key;
        values[size] = value;
    }
    size++;
}

void sort() {
    // Sort method is defined.
    int smallest = 0;
    for (int i = 0; i < size - 1; i++) {
        for (int j = i; j < size; j++) {
            if (keys[j] < keys[smallest])
                smallest = j;
        }
        swap(keys[i], keys[smallest]);
        swap(values[i], values[smallest]);
    }
}

void print() {
    for (int i = 0; i < size; i++)
        cout << "{" << keys[i] << ", " << values[i] << "}" << endl;
}
};

int main() {
    Dictionary<char*, char*>* dict = new Dictionary<char*, char*>(10);
    dict->print();
    dict->add("apple", "fruit");
    dict->add("banana", "fruit");
    dict->add("dog", "animal");
    dict->print();

    Dictionary<int, char*>* dict_specialized = new Dictionary<int, char*>(10);
    dict_specialized->print();
    dict_specialized->add(100, "apple");
    dict_specialized->add(101, "banana");
    dict_specialized->add(103, "dog");
    dict_specialized->add(89, "cat");
    dict_specialized->print();
    dict_specialized->sort();
    cout << endl << "Sorted list:" << endl;
    dict_specialized->print();
}
}

```

```

{apple, fruit}
{banana, fruit}
{dog, animal}
{100, apple}
{101, banana}
{103, dog}
{89, cat}

Sorted list:
{89, cat}
{100, apple}
{101, banana}
{103, dog}

```

Modelos e resolução de nome

25/03/2020 • 3 minutes to read • [Edit Online](#)

Em definições de modelo, há três tipos de nomes.

- Nomes declarados localmente, incluindo o nome do próprio modelo e alguns nomes declarados na definição de modelo.
- Nomes de escopo delimitador fora da definição de modelo.
- Nomes que dependem de alguma forma dos argumentos do modelo, conhecidos como nomes dependentes.

Embora os primeiros dois nomes também pertençam aos escopos de classe e função, regras especiais para a resolução de nomes são necessárias nas definições de modelo para lidar com a complexidade adicional de nomes dependentes. Isso é porque o compilador não sabe muito sobre esses nomes até que o modelo instanciado, porque podem ser tipos completamente diferentes dependendo dos argumentos de modelo usados. Os nomes não dependentes são pesquisados de acordo com as regras comuns e no momento da definição do modelo. Esses nomes, sendo independentes dos argumentos de modelo, são procurados uma vez para todas as especializações de modelo. Os nomes dependentes não são pesquisados até que o modelo seja instanciado e são pesquisados separadamente para cada especialização.

Um tipo é dependente quando depende dos argumentos do modelo. Especificamente, um tipo é dependente quando é:

- O próprio argumento do modelo:

```
T
```

- Um nome qualificado com uma qualificação que inclui um tipo dependente:

```
T::myType
```

- Um nome qualificado se a parte não qualificada identifica um tipo dependente:

```
N::T
```

- Um tipo const ou volatile cujo tipo de base é um tipo dependente:

```
const T
```

- Um ponteiro, uma referência, uma matriz, ou um tipo de ponteiro de função baseado em um tipo dependente:

```
T *, T &, T [10], T (*)()
```

- Uma matriz cujo tamanho é baseado em um parâmetro do modelo:

```
template <int arg> class X {  
    int x[arg] ; // dependent type  
}
```

- um tipo de modelo construído de um parâmetro do modelo:

```
T<int>, MyTemplate<T>
```

Dependência de tipo e dependência de valor

Os nomes e as expressões dependentes de um parâmetro de modelo são categorizados como dependentes de tipo ou de valor, dependendo se o parâmetro de modelo for um parâmetro de tipo ou um parâmetro de valor. Além disso, os identificadores declarados em um modelo com um tipo dependente do argumento de modelo são considerados dependentes de valor, pois é um tipo de integral ou de enumeração inicializado com uma expressão dependente de valor.

As expressões dependentes de tipo e de valor são expressões que envolvem variáveis que são dependentes de tipo ou de valor. Essas expressões podem ter a semântica diferente, dependendo dos parâmetros usados para o modelo.

Confira também

[Modelos](#)

Resolução de nome para tipos dependentes

02/09/2020 • 3 minutes to read • [Edit Online](#)

Use `typename` para nomes qualificados em definições de modelo para informar ao compilador que o nome qualificado fornecido identifica um tipo. Para obter mais informações, consulte [TypeName](#).

```
// template_name_resolution1.cpp
#include <stdio.h>
template <class T> class X
{
public:
    void f(typename T::myType* mt) {}
};

class Yarg
{
public:
    struct myType { };
};

int main()
{
    X<Yarg> x;
    x.f(new Yarg::myType());
    printf("Name resolved by using typename keyword.");
}
```

Name resolved by using typename keyword.

A pesquisa de nome para nomes dependentes examina nomes do contexto da definição do modelo — no exemplo a seguir, esse contexto encontraria `myFunction(char)` — e o contexto da instanciação do modelo. No exemplo a seguir, o modelo é instanciado em Main; Portanto, o `MyNamespace::myFunction` é visível a partir do ponto de instanciação e é escolhido como a melhor correspondência. Se `MyNamespace::myFunction` fosse renomeado, `myFunction(char)` seria chamado.

Todos os nomes são resolvidos como se fossem nomes dependentes. Entretanto, recomendamos que você use nomes totalmente qualificados se houver qualquer conflito possível.

```

// template_name_resolution2.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

void myFunction(char)
{
    cout << "Char myFunction" << endl;
}

template <class T> class Class1
{
public:
    Class1(T i)
    {
        // If replaced with myFunction(1), myFunction(char)
        // will be called
        myFunction(i);
    }
};

namespace MyNamespace
{
    void myFunction(int)
    {
        cout << "Int MyNamespace::myFunction" << endl;
    }
};

using namespace MyNamespace;

int main()
{
    Class1<int>* c1 = new Class1<int>(100);
}

```

Saída

```
Int MyNamespace::myFunction
```

Desambiguação de modelo

O Visual Studio 2012 impõe as regras padrão do C++ 98/03/11 para a desambiguidade com a palavra-chave "template". No exemplo a seguir, o Visual Studio 2010 aceitaria as linhas não conformes e as linhas em conformidade. O Visual Studio 2012 aceita apenas as linhas em conformidade.

```

#include <iostream>
#include <ostream>
#include <typeinfo>
using namespace std;

template <typename T> struct Allocator {
    template <typename U> struct Rebind {
        typedef Allocator<U> Other;
    };
};

template <typename X, typename AY> struct Container {
    #if defined(NONCONFORMANT)
        typedef typename AY::Rebind<X>::Other AX; // nonconformant
    #elif defined(CONFORMANT)
        typedef typename AY::template Rebind<X>::Other AX; // conformant
    #else
        #error Define NONCONFORMANT or CONFORMANT.
    #endif
};

int main() {
    cout << typeid(Container<int, Allocator<float>>::AX).name() << endl;
}

```

A conformidade com as regras de desambiguação é necessária porque, por padrão, o C++ presume que `AY::Rebind` não seja um modelo e, assim, o compilador interpreta o seguinte “`<`” como menor que. Ele precisa saber que `Rebind` é um modelo para poder analisar corretamente “`<`” como um colchete angular.

Confira também

[Resolução de nomes](#)

Resolução de nome para nomes declarados localmente

25/03/2020 • 5 minutes to read • [Edit Online](#)

O nome de modelo propriamente dito pode ser referenciado com ou sem os argumentos de modelo. No escopo de um modelo de classe, o nome propriamente dito se refere ao modelo. No escopo de uma especialização ou de uma especialização parcial de modelo, o nome sozinho se refere à especialização ou à especialização parcial. Outras especializações ou especializações parciais do modelo também podem ser referenciadas, com os argumentos de modelo apropriados.

Exemplo

O código a seguir mostra que o nome do modelo de classe A é interpretado de maneira diferente no escopo de uma especialização ou especialização parcial.

```
// template_name_resolution3.cpp
// compile with: /c
template <class T> class A {
    A* a1;    // A refers to A<T>
    A<int>* a2; // A<int> refers to a specialization of A.
    A<T*>* a3; // A<T*> refers to the partial specialization A<T*>.
};

template <class T> class A<T*> {
    A* a4; // A refers to A<T*>.
};

template<> class A<int> {
    A* a5; // A refers to A<int>.
};
```

Exemplo

Em caso de conflito de nome entre um parâmetro de modelo e outro objeto, o parâmetro de modelo pode ou não ser oculto. As regras a seguir ajudam a determinar a precedência.

O parâmetro de modelo está no escopo desde o ponto onde aparece primeiro até o final do modelo de classe ou de função. Se o nome aparece novamente na lista de argumentos do modelo ou na lista de classes base, ele faz referência ao mesmo tipo. No C++ padrão, nenhum outro nome idêntico ao parâmetro de modelo pode ser declarado no mesmo escopo. Uma extensão da Microsoft permite que o parâmetro de modelo seja redefinido no escopo do modelo. O exemplo a seguir mostra o uso do parâmetro de modelo na especificação da base de um modelo de classe.

```

// template_name_resolution4.cpp
// compile with: /EHsc
template <class T>
class Base1 {};

template <class T>
class Derived1 : Base1<T> {};

int main() {
    // Derived1<int> d;
}

```

Exemplo

Ao definir as funções de membro de um modelo fora do modelo de classe, é possível usar um nome de parâmetro de modelo diferente. Se a definição de função de membro de modelo usar, para o parâmetro de modelo, um nome diferente do usado pela declaração e o nome usado na definição entrar em conflito com outro membro da declaração, o membro na declaração de modelo terá precedência.

```

// template_name_resolution5.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

template <class T> class C {
public:
    struct Z {
        Z() { cout << "Z::Z()" << endl; }
    };
    void f();
};

template <class Z>
void C<Z>::f() {
    // Z refers to the struct Z, not to the template arg;
    // Therefore, the constructor for struct Z will be called.
    Z z;
}

int main() {
    C<int> c;
    c.f();
}

```

```

Z::Z()

```

Exemplo

Ao definir uma função de modelo ou uma função de membro fora do namespace no qual o modelo foi declarado, o argumento de modelo tem precedência sobre os nomes de outros membros do namespace.

```

// template_name_resolution6.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

namespace NS {
    void g() { cout << "NS::g" << endl; }

    template <class T> struct C {
        void f();
        void g() { cout << "C<T>::g" << endl; }
    };
}

template <class T>
void NS::C<T>::f() {
    g(); // C<T>::g, not NS::g
}

int main() {
    NS::C<int> c;
    c.f();
}

```

C<T>::g

Exemplo

Nas definições que estão fora da declaração de classe de modelo, se uma classe de modelo tiver uma classe base que não dependa de um argumento de modelo e se a classe base ou um de seus membros tiver o mesmo nome que um argumento de modelo, a classe base ou o nome de membro ocultarão o argumento de modelo.

```

// template_name_resolution7.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

struct B {
    int i;
    void print() { cout << "Base" << endl; }
};

template <class T, int i> struct C : public B {
    void f();
};

template <class B, int i>
void C<B, i>::f() {
    B b; // Base class b, not template argument.
    b.print();
    i = 1; // Set base class's i to 1.
}

int main() {
    C<int, 1> c;
    c.f();
    cout << c.i << endl;
}

```

Base

1

Confira também

[Resolução de nomes](#)

Resolução de sobrecarga das chamadas de modelo de função

25/03/2020 • 2 minutes to read • [Edit Online](#)

Um modelo de função pode sobrecarregar funções de fora do modelo com o mesmo nome. Nesse cenário, as chamadas de função são resolvidas primeiro com o uso de dedução de argumentos do modelo para criar uma instância do modelo de função com uma especialização exclusiva. Se a dedução de argumentos do modelo falhar, as outras sobrecargas de função são consideradas resolver a chamada. Essas outras sobrecargas, também conhecidas como conjunto de candidatas, incluem funções fora do modelo e outros modelos de função com instâncias. Se a dedução de argumentos do modelo for bem sucedida, a função gerada será comparada com as outras funções para determinar a melhor correspondência, de acordo com as regras de resolução de sobrecarga. Para obter mais informações, consulte [sobrecarga de função](#).

Exemplo

Se uma função fora do modelo for uma correspondência igualmente boa a uma função do modelo, a função fora do modelo será escolhida (a menos que os argumentos do modelo sejam especificados explicitamente), como na chamada `f(1, 1)` no exemplo a seguir.

```
// template_name_resolution9.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

void f(int, int) { cout << "f(int, int)" << endl; }
void f(char, char) { cout << "f(char, char)" << endl; }

template <class T1, class T2>
void f(T1, T2)
{
    cout << "void f(T1, T2)" << endl;
}

int main()
{
    f(1, 1);    // Equally good match; choose the nontemplate function.
    f('a', 1); // Chooses the template function.
    f<int, int>(2, 2); // Template arguments explicitly specified.
}
```

```
f(int, int)
void f(T1, T2)
void f(T1, T2)
```

Exemplo

O exemplo a seguir mostra que a função de modelo com correspondência exata é preferida se a função fora do modelo requeira uma conversão.

```
// template_name_resolution10.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

void f(int, int) { cout << "f(int, int)" << endl; }

template <class T1, class T2>
void f(T1, T2)
{
    cout << "void f(T1, T2)" << endl;
};

int main()
{
    long l = 0;
    int i = 0;
    // Call the template function f(long, int) because f(int, int)
    // would require a conversion from long to int.
    f(l, i);
}
```

```
void f(T1, T2)
```

Confira também

[Resolução de nomes](#)

[typename](#)

Organização do código-fonte (modelos do C++)

02/09/2020 • 7 minutes to read • [Edit Online](#)

Ao definir um modelo de classe, você deve organizar o código-fonte de tal forma que as definições de membro sejam visíveis para o compilador quando ele precisar delas. Você tem a opção de usar o *modelo de inclusão* ou o *modelo de instanciação explícita*. No modelo de inclusão, você inclui as definições de membro em cada arquivo que usa um modelo. Essa abordagem é a mais simples e oferece a máxima flexibilidade em termos dos tipos concretos que podem ser usados com seu modelo. A desvantagem é que pode aumentar o tempo de compilação. O impacto pode ser significativo se um projeto e/ou os próprios arquivos incluídos forem grandes. Com a abordagem de instanciação explícita, o próprio modelo instancia as classes concretas ou membros de classe para tipos específicos. Essa abordagem pode acelerar os tempos de compilação, mas limita o uso somente para as classes que o implementador de modelo habilitou antecipadamente. Em geral, é recomendável que você use o modelo de inclusão, a menos que os tempos de compilação se tornem um problema.

Segundo plano

Os modelos não são como classes comuns no sentido de que o compilador não gera o código de objeto para um modelo ou qualquer um dos seus membros. Não há nada a ser gerado até que o modelo seja instanciado com tipos concretos. Quando o compilador encontra uma instanciação de modelo, como `MyClass<int> mc;`, e ainda não existe nenhuma classe com essa assinatura, ele gera uma nova classe. Ele também tenta gerar código para todas as funções membro que são usadas. Se essas definições estiverem em um arquivo que não esteja #incluído, direta ou indiretamente, no arquivo .cpp que está sendo compilado, o compilador não poderávê-los. Do ponto de vista do compilador, isso não é necessariamente um erro porque as funções podem ser definidas em outra unidade de tradução, que serão encontradas pelo vinculador. Caso o vinculador não encontre esse código, ele gerará um erro *externo não resolvido*.

O modelo de inclusão

A maneira mais simples e mais comum de tornar visíveis as definições de modelo em uma unidade de tradução é colocando-as no próprio arquivo de cabeçalho. Todo arquivo .cpp que usa o modelo simplesmente tem que #incluir o cabeçalho. Essa é a abordagem usada na Biblioteca Padrão.

```

#ifndef MYARRAY
#define MYARRAY
#include <iostream>

template<typename T, size_t N>
class MyArray
{
    T arr[N];
public:
    // Full definitions:
    MyArray(){}
    void Print()
    {
        for (const auto v : arr)
        {
            std::cout << v << " , ";
        }
    }
    T& operator[](int i)
    {
        return arr[i];
    }
};

#endif

```

Com essa abordagem, o compilador tem acesso à definição completa do modelo e pode instanciar modelos sob demanda para qualquer tipo. É simples e relativamente fácil de manter. No entanto, o modelo de inclusão tem um custo em termos de tempo de compilação. Esse custo pode ser significativo em programas maiores, especialmente se o cabeçalho do modelo em si #incluir outros cabeçalhos. Cada `.cpp` arquivo que #includes cabeçalho receberá sua própria cópia dos modelos de função e todas as definições. O vinculador geralmente conseguirá arrumar tudo para que você não acabe com várias definições para uma função, mas realizar esse trabalho leva tempo. Em programas menores, esse tempo de compilação a mais provavelmente não é significativo.

O modelo de instanciação explícita

Se o modelo de inclusão não for viável para seu projeto e você souber claramente o conjunto de tipos que será usado para instanciar um modelo, poderá separar o código do modelo em um `.h` `.cpp` arquivo e, no `.cpp` arquivo, instanciar explicitamente os modelos. Isso fará com que o código de objeto que o compilador verá quando encontrar instanciações do usuário seja gerado.

Uma instanciação explícita é criada, usando o modelo de palavra-chave seguido da assinatura da entidade que você deseja instanciar. Pode ser um tipo ou membro. Caso você instancie explicitamente um tipo, todos os membros serão instanciados.

```
template MyArray<double, 5>;
```

```

//MyArray.h
#ifndef MYARRAY
#define MYARRAY

template<typename T, size_t N>
class MyArray
{
    T arr[N];
public:
    MyArray();
    void Print();
    T& operator[](int i);
};

#endif

//MyArray.cpp
#include <iostream>
#include "MyArray.h"

using namespace std;

template<typename T, size_t N>
MyArray<T,N>::MyArray(){}
template<typename T, size_t N>
void MyArray<T,N>::Print()
{
    for (const auto v : arr)
    {
        cout << v << " ";
    }
    cout << endl;
}

template MyArray<double, 5>;template MyArray<string, 5>;

```

No exemplo anterior, as instanciações explícitas estão na parte inferior do arquivo .cpp. Um `MyArray` só pode ser usado para `double` `String` tipos ou.

NOTE

Em C++ 11, a `export` palavra-chave foi preferida no contexto de definições de modelo. Em termos práticos, isso tem pouco impacto porque a maioria dos compiladores nunca foi compatível com ela.

Tratamento de Evento

25/03/2020 • 2 minutes to read • [Edit Online](#)

A manipulação de eventos é basicamente suportada C++ para classes com (classes que implementam objetos com, normalmente usando classes ATL ou o atributo `coclass`). Para obter mais informações, consulte [manipulação de eventos em com](#).

A manipulação de eventos também tem suporte para classes nativas de C++ (classes C++ que não implementam objetos COM). Entretanto, esse suporte é preterido e será removido em uma versão futura. Para obter mais informações, consulte [manipulação de eventos C++ em nativo](#).

A manipulação de eventos tem suporte ao uso único e multithread e protege os dados contra acessos simultâneos multithread. Ele também permite que você gere subclasses das classes de origem ou do receptor de eventos e tem suporte a fontes/recebimentos de eventos na classe derivada.

O compilador C++ da Microsoft inclui atributos e palavras-chave para declarar eventos e manipuladores de eventos. Os atributos de eventos e as palavras-chave podem ser usados em programas CLR e em programas nativos C++.

TÓPICO	DESCRIÇÃO
<code>event_source</code>	Cria uma origem de evento.
<code>event_receiver</code>	Cria um receptor de eventos (coletor).
<code>_event</code>	Declara um evento.
<code>_raise</code>	Enfatiza o site de chamada de um evento.
<code>_hook</code>	Associa um método de manipulador a um evento.
<code>_unhook</code>	Dissocia um método de manipulador de um evento.

Confira também

[Referência da linguagem C++](#)

[Palavras-chave](#)

__event

02/09/2020 • 6 minutes to read • [Edit Online](#)

Declara um evento.

Sintaxe

```
__event method-declarator;  
__event __interface interface-specifier;  
__event member-declarator;
```

Comentários

A palavra-chave `__event` pode ser aplicada a uma declaração de método, uma declaração de interface ou uma declaração de membro de dados. No entanto, você não pode usar a `__event` palavra-chave para qualificar um membro de uma classe aninhada.

Dependendo se sua fonte de evento e receptor são C++ Nativo, COM ou gerenciados (.NET Framework), você pode usar as seguintes construções como eventos:

C++ NATIVO	COM	GERENCIADO (.NET FRAMEWORK)
Método	—	method
—	interface	—
—	—	membro de dados

Use `__hook` em um receptor de eventos para associar um manipulador a um método de evento. Observe que, depois de criar um evento com a `__event` palavra-chave, todos os manipuladores de eventos, posteriormente conectados a esse evento, serão chamados quando o evento for chamado.

Uma `__event` declaração de método não pode ter uma definição; uma definição é gerada implicitamente, portanto, o método de evento pode ser chamado como se fosse um método comum.

NOTE

Uma classe ou um struct modelo não podem conter eventos.

Eventos nativos

Os eventos nativos são métodos. O tipo de retorno é normalmente `HRESULT` ou `void`, mas pode ser qualquer tipo integral, incluindo um `enum`. Quando um evento usa um tipo de retorno integral, uma condição de erro é definida quando um manipulador de eventos retorna um valor diferente de zero. Nesse caso, o evento sendo gerado chama os outros delegados.

```
// Examples of native C++ events:  
__event void OnDoubleClick();  
__event HRESULT OnClick(int* b, char* s);
```

Consulte [manipulação de eventos em C++ nativo](#) para código de exemplo.

Eventos COM

Os eventos COM são interfaces. Os parâmetros de um método em uma interface de origem de evento devem estar *em* parâmetros (mas isso não é rigorosamente imposto), pois um parâmetro *out* não é útil quando o multicast. Um aviso de nível 1 será emitido se você usar um parâmetro *out*.

O tipo de retorno é normalmente HRESULT ou `void`, mas pode ser qualquer tipo integral, incluindo `enum`. Quando um evento usa um tipo de retorno integral e um manipulador de eventos retorna uma condição de erro, nesse caso, o evento sendo gerado aborta as chamadas para outros delegados. Observe que o compilador marcará automaticamente uma interface de origem de evento como uma [origem](#) na IDL gerada.

A palavra-chave `__interface` sempre é necessária após `__event` para uma origem de evento com.

```
// Example of a COM event:  
__event __interface IEvent1;
```

Consulte [manipulação de eventos em com](#) para obter o código de exemplo.

Eventos gerenciadas

Para obter informações sobre eventos de codificação na nova sintaxe, consulte [evento](#).

Os eventos gerenciados são membros de dados ou métodos. Quando usado com um evento, o tipo de retorno de um delegado deve ser compatível com o [Common Language Specification](#). O tipo de retorno do manipulador de eventos deve corresponder ao tipo de retorno do delegado. Para obter mais informações sobre delegados, consulte [delegados e eventos](#). Se um evento gerenciado for um membro de dados, seu tipo deve ser um ponteiro para um delegado.

No .NET Framework, você pode tratar um membro de dados como um método (ou seja, o método `Invoke` do delegado correspondente). Você deve predefinir o tipo de delegado para declarar um membro de dados do evento gerenciado. Por outro lado, um método de evento gerenciado define implicitamente o delegado gerenciado correspondente, se ele ainda não tiver sido definido. Por exemplo, você pode declarar um valor do evento como `OnClick` como um evento a seguir:

```
// Examples of managed events:  
__event ClickEventHandler* OnClick; // data member as event  
__event void OnClick(String* s); // method as event
```

Ao declarar implicitamente um evento gerenciado, você pode especificamente adicionar e remover acessadores que serão chamados quando manipuladores de eventos forem adicionados ou removidos. Você também pode definir o método que chama (gera) o evento de fora da classe.

Exemplo: eventos nativos

```
// EventHandling_Native_Event.cpp
// compile with: /c
[event_source(native)]
class CSource {
public:
    __event void MyEvent(int nValue);
};
```

Exemplo: eventos COM

```
// EventHandling_COM_Event.cpp
// compile with: /c
#define _ATL_ATTRIBUTES 1
#include <atlbase.h>
#include <atlcom.h>

[ module(dll, name="EventSource", uuid="6E46B59E-89C3-4c15-A6D8-B8A1CEC98830") ];

[ dual, uuid("00000000-0000-0000-0000-000000000002") ]
__interface IEventSource {
    [id(1)] HRESULT MyEvent();
};

[ coclass, uuid("00000000-0000-0000-0000-000000000003"), event_source(com) ]
class CSource : public IEventSource {
public:
    __event __interface IEventSource;
    HRESULT FireEvent() {
        __raise MyEvent();
        return S_OK;
    }
};
```

Confira também

[Palavras-chave](#)

[Manipulação de eventos](#)

[event_source](#)

[event_receiver](#)

[__hook](#)

[__unhook](#)

[__raise](#)

__hook

02/09/2020 • 5 minutes to read • [Edit Online](#)

Associa um método de manipulador a um evento.

Sintaxe

```
long __hook(
    &SourceClass::EventMethod,
    source,
    &ReceiverClass::HandlerMethod
    [, receiver = this]
);
long __hook(
    interface,
    source
);
```

parâmetros

&SourceClass::EventMethod

Um ponteiro para o método de evento ao qual você engancha o método do manipulador de eventos:

- Eventos C++ nativos: *SourceClass* é a classe de origem do evento e *EventMethod* é o evento.
- Eventos COM: *SourceClass* é a interface de origem do evento e *EventMethod* é um dos seus métodos.
- Eventos gerenciados: *SourceClass* é a classe de origem do evento e *EventMethod* é o evento.

interface

O nome da interface que está sendo conectada ao *receptor*, somente para receptores de eventos com nos quais o parâmetro *layout_dependent* do atributo *event_receiver* é `true`.

source

Um ponteiro para uma instância da origem do evento. Dependendo do código `type` especificado em `event_receiver`, a *origem* pode ser uma das seguintes:

- Um ponteiro nativo do objeto de origem do evento.
- Um `IUnknown` ponteiro baseado em (fonte com).
- Um ponteiro gerenciado do objeto (para eventos gerenciados).

&ReceiverClass::HandlerMethod

Um ponteiro para o método do manipulador de eventos a ser enganchado a um evento. O manipulador é especificado como um método de uma classe ou uma referência para o mesmo; Se você não especificar o nome da classe, `__hook` o assumirá que a classe é a qual ela será chamada.

- Eventos C++ nativos: *ReceiverClass* é a classe receptora de evento e `HandlerMethod` é o manipulador.
- Eventos COM: *ReceiverClass* é a interface do receptor de eventos e `HandlerMethod` é um de seus manipuladores.
- Eventos gerenciados: *ReceiverClass* é a classe receptora de evento e `HandlerMethod` é o manipulador.

distância

Adicional Um ponteiro para uma instância da classe receptor do evento. Se você não especificar um destinatário, o padrão será a classe ou a estrutura do receptor em que `__hook` é chamada.

Uso

Pode ser o uso em qualquer escopo da função, incluindo o principal, fora da classe do receptor de eventos.

Comentários

Use a função intrínseca `__hook` em um receptor de eventos para associar ou vincular um método de manipulador a um método de evento. O manipulador especificado é chamado quando a origem gera o evento especificado. Você pode enganchar vários manipuladores a um único evento, ou enganchar vários eventos a um único manipulador.

Há duas formas de `__hook`. Você pode usar o primeiro formulário (quatro argumentos) na maioria dos casos, especificamente, para receptores de eventos COM, nos quais o parâmetro *layout_dependent* do atributo `event_receiver` é `false`.

Nesses casos você não precisa enganchar todos os métodos em uma interface antes de acionar um evento em um dos métodos; somente a manipulação de método do evento precisa ser enganchado. Você pode usar a segunda forma (de dois argumentos) `__hook` somente para um receptor de evento com no qual `layout_dependent = true`.

`__hook` Retorna um valor longo. Um valor de retorno diferente de zero indica que ocorreu um erro (eventos gerenciados lançam uma exceção).

O compilador verifica a existência de um evento e se a assinatura do evento concorda com a assinatura de delegação.

Com exceção de eventos COM, `__hook` e `__unhook` pode ser chamado fora do receptor de eventos.

Uma alternativa ao uso `__hook` do é usar o operador `+ =`.

Para obter informações sobre como codificar eventos gerenciados na nova sintaxe, consulte [evento](#).

NOTE

Uma classe ou um struct modelo não podem conter eventos.

Exemplo

Consulte [manipulação de eventos em C++ nativo](#) e [manipulação de eventos em com](#) para obter exemplos.

Confira também

[Palavras-chave](#)

[Manipulação de eventos](#)

[event_source](#)

[event_receiver](#)

[__unhook](#)

[__raise](#)

__raise

02/09/2020 • 2 minutes to read • [Edit Online](#)

Enfatiza o site de chamada de um evento.

Sintaxe

```
__raise method-declarator;
```

Comentários

No código gerenciado, um evento só pode ser acionado de dentro da classe em que é definido. Consulte o [evento](#) para obter mais informações.

A palavra-chave `__raise` faz com que um erro seja emitido se você chamar um não evento.

NOTE

Uma classe ou um struct modelo não podem conter eventos.

Exemplo

```
// EventHandlingRef_raise.cpp
struct E {
    __event void func1();
    void func1(int) {}

    void func2() {}

    void b() {
        __raise func1();
        __raise func1(1); // C3745: 'int Event::bar(int)':
                          // only an event can be 'raised'
        __raise func2(); // C3745
    }
};

int main() {
    E e;
    __raise e.func1();
    __raise e.func1(1); // C3745
    __raise e.func2(); // C3745
}
```

Confira também

[Palavras-chave](#)

[Manipulação de eventos](#)

[Extensões de componentes para plataformas de runtime](#)

__unhook

02/09/2020 • 5 minutes to read • [Edit Online](#)

Dissocia um método de manipulador de um evento.

Sintaxe

```
long __unhook(
    &SourceClass::EventMethod,
    source,
    &ReceiverClass::HandlerMethod
    [, receiver = this]
);
long __unhook(
    interface,
    source
);
long __unhook(
    source
);
```

parâmetros

`*&***SourceClass` :: `EventMethod` Um ponteiro para o método de evento do qual você desengancha o método do manipulador de eventos:

- Eventos C++ nativos: `SourceClass` é a classe de origem do evento e `EventMethod` é o evento.
- Eventos COM: `SourceClass` é a interface de origem do evento e `EventMethod` é um dos seus métodos.
- Eventos gerenciados: `SourceClass` é a classe de origem do evento e `EventMethod` é o evento.

interface

O nome da interface que está sendo desconectado do `receptor`, somente para receptores de eventos com nos quais o parâmetro `layout_dependent` do atributo `event_receiver` é `true`.

source

Um ponteiro para uma instância da origem do evento. Dependendo do código `type` especificado em `event_receiver`, a `origem` pode ser uma das seguintes:

- Um ponteiro nativo do objeto de origem do evento.
- Um `IUnknown` ponteiro baseado em (fonte com).
- Um ponteiro gerenciado do objeto (para eventos gerenciados).

`*&***ReceiverClass` :: `HandlerMethod` Um ponteiro para o método manipulador de eventos a ser desvinculado de um evento. O manipulador é especificado como um método de uma classe ou uma referência para o mesmo; Se você não especificar o nome da classe, `__unhook` o assumirá que a classe é a qual ela será chamada.

- Eventos C++ nativos: `ReceiverClass` é a classe receptora de evento e `HandlerMethod` é o manipulador.
- Eventos COM: `ReceiverClass` é a interface do receptor de eventos e `HandlerMethod` é um de seus manipuladores.
- Eventos gerenciados: `ReceiverClass` é a classe receptora de evento e `HandlerMethod` é o manipulador.

Receiver(opcional) um ponteiro para uma instância da classe receptor do evento. Se você não especificar um destinatário, o padrão será a classe ou a estrutura do receptor em que `__unhook` é chamada.

Uso

Pode ser o uso em qualquer escopo da função, incluindo o principal, fora da classe do receptor de eventos.

Comentários

Use a função intrínseca `__unhook` em um receptor de eventos para dissociar ou "desvincular" um método de manipulador de um método de evento.

Há três formas de `__unhook`. Você pode usar o primeiro formulário (quatro argumentos) na maioria dos casos. Você pode usar a segunda forma (de dois argumentos) `__unhook` apenas para um receptor de evento com; isso desvincula toda a interface de evento. Você pode usar o terceiro formato (um argumento) para desenganchar todos os representantes da origem especificada.

Um valor de retorno diferente de zero indica que ocorreu um erro (eventos gerenciados lançarão uma exceção).

Se você chamar `__unhook` em um evento e manipulador de eventos que ainda não foram conectados, ele não terá nenhum efeito.

Em tempo de compilação, o compilador verifica se o evento existe e faz a verificação do tipo de parâmetro com o manipulador especificado.

Com exceção de eventos COM, `__hook` e `__unhook` pode ser chamado fora do receptor de eventos.

Uma alternativa ao uso `__unhook` do é usar o operador-=.

Para obter informações sobre como codificar eventos gerenciados na nova sintaxe, consulte [evento](#).

NOTE

Uma classe ou um struct modelo não podem conter eventos.

Exemplo

Consulte [manipulação de eventos em C++ nativo](#) e [manipulação de eventos em com](#) para obter exemplos.

Confira também

[Palavras-chave](#)

[event_source](#)

[event_receiver](#)

[__event](#)

[__hook](#)

[__raise](#)

Tratamento de eventos em C++ nativo

25/03/2020 • 3 minutes to read • [Edit Online](#)

Na manipulação C++ de eventos nativos, você configura uma origem de evento e um receptor de evento usando os atributos `EVENT_SOURCE` e `event_receiver`, respectivamente, especificando `type = native`. Esses atributos as permitem que as classes às quais eles são aplicados disparem eventos e manipulem eventos em um contexto nativo, e não de COM.

Declarando eventos

Em uma classe de origem de evento, use a palavra-chave `_event` em uma declaração de método para declarar o método como um evento. Certifique-se de declarar o método, mas não defina-o; fazer isso gerará um erro do compilador, pois o compilador define o método implicitamente quando ele é transformado em um evento. Os eventos nativos podem ser métodos com zero ou mais parâmetros. O tipo de retorno pode ser `void` ou qualquer tipo integral.

Definindo manipuladores de eventos

Em uma classe de receptor de evento, você define manipuladores de eventos, que são métodos com assinaturas (tipos de retorno, convenções de chamada e argumentos) que correspondem ao evento que eles manipularão.

Vinculando manipuladores de eventos a eventos

Também em uma classe receptora de eventos, você usa a função intrínseca `_hook` para associar eventos a manipuladores de eventos e `_unhook` para dissociar eventos de manipuladores de eventos. Você pode vincular diversos eventos a um manipulador ou vincular diversos manipuladores a um evento.

Acionando eventos

Para disparar um evento, simplesmente chame o método declarado como um evento na classe da origem do evento. Se houver manipuladores vinculados ao evento, eles serão chamados.

Código nativo do evento C++

O exemplo de código a seguir mostra como disparar um evento no C++ nativo. Para compilar e executar o exemplo, consulte os comentários no código.

Exemplo

Código

```

// evh_native.cpp
#include <stdio.h>

[event_source(native)]
class CSource {
public:
    __event void MyEvent(int nValue);
};

[event_receiver(native)]
class CReceiver {
public:
    void MyHandler1(int nValue) {
        printf_s("MyHandler1 was called with value %d.\n", nValue);
    }

    void MyHandler2(int nValue) {
        printf_s("MyHandler2 was called with value %d.\n", nValue);
    }

    void hookEvent(CSource* pSource) {
        __hook(&CSource::MyEvent, pSource, &CReceiver::MyHandler1);
        __hook(&CSource::MyEvent, pSource, &CReceiver::MyHandler2);
    }

    void unhookEvent(CSource* pSource) {
        __unhook(&CSource::MyEvent, pSource, &CReceiver::MyHandler1);
        __unhook(&CSource::MyEvent, pSource, &CReceiver::MyHandler2);
    }
};

int main() {
    CSource source;
    CReceiver receiver;

    receiver.hookEvent(&source);
    __raise source.MyEvent(123);
    receiver.unhookEvent(&source);
}

```

Saída

```

MyHandler2 was called with value 123.
MyHandler1 was called with value 123.

```

Confira também

[Manipulação de eventos](#)

Tratamento de eventos em COM

02/09/2020 • 7 minutes to read • [Edit Online](#)

Na manipulação de eventos COM, você configura uma origem de evento e um receptor de evento usando os atributos `EVENT_SOURCE` e `event_receiver`, respectivamente, especificando `type = com`. Esses atributos injetam o código apropriado para interfaces personalizadas, duais e de expedição a fim de permitir que as classes às quais são aplicados acessem eventos e manipulem eventos por meio de pontos de conexão COM.

Declarando eventos

Em uma classe de origem de evento, use a palavra-chave `_event` em uma declaração de interface para declarar os métodos da interface como eventos. Os eventos dessa interface são acionados quando você os chama como métodos da interface. Métodos em interfaces de evento podem ter zero ou mais parâmetros (que devem estar todos *em* parâmetros). O tipo de retorno pode ser `void` ou qualquer tipo integral.

Definindo manipuladores de eventos

Em uma classe de receptor de evento, você define manipuladores de eventos, que são métodos com assinaturas (tipos de retorno, convenções de chamada e argumentos) que correspondem ao evento que eles manipularão. Para eventos COM, as convenções de chamada não precisam corresponder; consulte [eventos com dependentes de layout](#) abaixo para obter detalhes.

Vinculando manipuladores de eventos a eventos

Também em uma classe receptora de eventos, você usa a função intrínseca `_hook` para associar eventos a manipuladores de eventos e `_unhook` para dissociar eventos de manipuladores de eventos. Você pode vincular diversos eventos a um manipulador ou vincular diversos manipuladores a um evento.

NOTE

Normalmente, há duas técnicas para permitir que um receptor de evento COM acesse definições de interface de origem de evento. O primeiro, conforme mostrado abaixo, é compartilhar um arquivo de cabeçalho comum. A segunda é usar `#import` com o `embedded_id1` qualificador de importação, para que a biblioteca do tipo de origem do evento seja gravada no arquivo. tlh com o código gerado pelo atributo preservado.

Acionando eventos

Para acionar um evento, basta chamar um método na interface declarada com a `_event` palavra-chave na classe de origem do evento. Se houver manipuladores vinculados ao evento, eles serão chamados.

Código de evento COM

O exemplo a seguir mostra como acionar um evento em uma classe COM. Para compilar e executar o exemplo, consulte os comentários no código.

```

// evh_server.h
#pragma once

[ dual, uuid("00000000-0000-0000-0000-000000000001") ]
__interface IEvents {
    [id(1)] HRESULT MyEvent([in] int value);
};

[ dual, uuid("00000000-0000-0000-0000-000000000002") ]
__interface IEventSource {
    [id(1)] HRESULT FireEvent();
};

class DECLSPEC_UUID("530DF3AD-6936-3214-A83B-27B63C7997C4") CSource;

```

Então o servidor:

```

// evh_server.cpp
// compile with: /LD
// post-build command: Regsvr32.exe /s evh_server.dll
#define _ATL_ATTRIBUTES 1
#include <atlbase.h>
#include <atlcom.h>
#include "evh_server.h"

[ module(dll, name="EventSource", uuid="6E46B59E-89C3-4c15-A6D8-B8A1CEC98830") ];

[coclass, event_source(com), uuid("530DF3AD-6936-3214-A83B-27B63C7997C4")]
class CSource : public IEventSource {
public:
    __event __interface IEvents;

    HRESULT FireEvent() {
        __raise MyEvent(123);
        return S_OK;
    }
};

```

Então o cliente:

```

// evh_client.cpp
// compile with: /link /OPT:NOREF
#define _ATL_ATTRIBUTES 1
#include <atlbase.h>
#include <atlcom.h>
#include <stdio.h>
#include "evh_server.h"

[ module(name="EventReceiver") ];

[ event_receiver(com) ]
class CReceiver {
public:
    HRESULT MyHandler1(int nValue) {
        printf_s("MyHandler1 was called with value %d.\n", nValue);
        return S_OK;
    }

    HRESULT MyHandler2(int nValue) {
        printf_s("MyHandler2 was called with value %d.\n", nValue);
        return S_OK;
    }

    void HookEvent(IEventSource* pSource) {
        __hook(&IEvents::MyEvent, pSource, &CReceiver::MyHandler1);
        __hook(&IEvents::MyEvent, pSource, &CReceiver::MyHandler2);
    }

    void UnhookEvent(IEventSource* pSource) {
        __unhook(&IEvents::MyEvent, pSource, &CReceiver::MyHandler1);
        __unhook(&IEvents::MyEvent, pSource, &CReceiver::MyHandler2);
    }
};

int main() {
    // Create COM object
    CoInitialize(NULL);
    {
        IEventSource* pSource = 0;
        HRESULT hr = CoCreateInstance(__uuidof(CSource), NULL, CLSCTX_ALL, __uuidof(IEventSource),
        (void **) &pSource);
        if (FAILED(hr)) {
            return -1;
        }

        // Create receiver and fire event
        CReceiver receiver;
        receiver.HookEvent(pSource);
        pSource->FireEvent();
        receiver.UnhookEvent(pSource);
    }
    CoUninitialize();
    return 0;
}

```

Saída

```

MyHandler1 was called with value 123.
MyHandler2 was called with value 123.

```

Eventos COM dependentes de layout

A dependência do layout só é um problema para a programação COM. Na manipulação de eventos nativos e

gerenciados, as assinaturas (tipo de retorno, convenção de chamada e argumentos) dos manipuladores devem corresponder aos respectivos eventos, mas os nomes dos manipuladores não precisam corresponder aos respectivos eventos.

No entanto, na manipulação de eventos COM, quando você define o parâmetro `layout_dependent` de `event_receiver` como `true`, a correspondência de nome e assinatura é imposta. Isso significa que os nomes e as assinaturas dos manipuladores no receptor de evento devem corresponder exatamente aos nomes e às assinaturas dos eventos aos quais estão vinculados.

Quando `layout_dependent` é definido como `false`, a Convenção de chamada e a classe de armazenamento (virtual, estática e assim por diante) podem ser misturadas e correspondidas entre o método de acionamento de evento e os métodos de conexão (seus delegados). É um pouco mais eficiente ter `layout_dependent = true`.

Por exemplo, suponha que `IEventSource` esteja definido para ter os seguintes métodos:

```
[id(1)] HRESULT MyEvent1([in] int value);
[id(2)] HRESULT MyEvent2([in] int value);
```

Pressuponha que a origem do evento tenha o seguinte formato:

```
[coclass, event_source(com)]
class CSource : public IEventSource {
public:
    __event __interface IEvents;

    HRESULT FireEvent() {
        MyEvent1(123);
        MyEvent2(123);
        return S_OK;
    }
};
```

Em seguida, no receptor do evento, qualquer manipulador vinculado a um método em `IEventSource` deve corresponder ao nome e à assinatura, como se segue:

```
[coclass, event_receiver(com, true)]
class CReceiver {
public:
    HRESULT MyEvent1(int nValue) { // name and signature matches MyEvent1
        ...
    }
    HRESULT MyEvent2(E c, char* pc) { // signature doesn't match MyEvent2
        ...
    }
    HRESULT MyHandler1(int nValue) { // name doesn't match MyEvent1 (or 2)
        ...
    }
    void HookEvent(IEventSource* pSource) {
        __hook(IFace, pSource); // Hooks up all name-matched events
                                // under layout_dependent = true
        __hook(&IFace::MyEvent1, pSource, &CReceive::MyEvent1); // valid
        __hook(&IFace::MyEvent2, pSource, &CSink::MyEvent2); // not valid
        __hook(&IFace::MyEvent1, pSource, &CSink:: MyHandler1); // not valid
    }
};
```

Confira também

[Manipulação de eventos](#)

Modificadores específicos da Microsoft

06/12/2019 • 2 minutes to read • [Edit Online](#)

Esta seção descreve extensões específicas da Microsoft para C++ nas seguintes áreas:

- [Com base em endereçamento](#), a prática de usar um ponteiro como uma base da qual outros ponteiros podem ser deslocados
- [Convenções de chamada de função](#)
- Atributos de classe de armazenamento estendidos declarados com a palavra-chave [__declspec](#)
- A palavra-chave [__w64](#)

palavras-chave específicas da Microsoft

Várias das palavras-chave específicas da Microsoft podem ser usadas para modificar declaradores para formar tipos derivados. Para obter mais informações sobre declaradores, consulte [declaradores](#).

PALAVRA-CHAVE	SIGNIFICADO	USADA PARA FORMAR TIPOS DERIVADOS?
__based	O nome que segue declara um deslocamento de 32 bits para a base de 32 bits contida na declaração.	Sim
__cdecl	O nome que segue usa as convenções de nomenclatura e chamada do C.	Sim
__declspec	O nome que segue especifica um atributo de classe de armazenamento específico da Microsoft.	Não
__fastcall	O nome que segue declara uma função que usa registros, quando disponíveis, em vez da pilha para passar argumentos.	Sim
__restrict	Semelhante a __declspec (restrict), mas para uso em variáveis.	Não
__stdcall	O nome que se segue especifica uma função que observa a convenção padrão de chamada.	Sim
__w64	Marca um tipo de dados como sendo maior em um compilador de 64 bits.	Não
__unaligned	Especifica que um ponteiro para um tipo ou outros dados não está alinhado.	Não

PALAVRA-CHAVE	SIGNIFICADO	USADA PARA FORMAR TIPOS DERIVADOS?
<code>_vectorcall</code>	O nome que segue declara uma função que usa registros, incluindo registros SSE, quando disponíveis, em vez da pilha para passar argumentos.	Sim

Consulte também

[Referência da linguagem C++](#)

Endereçamento com base em

25/03/2020 • 2 minutes to read • [Edit Online](#)

Esta seção inclui os tópicos a seguir:

- [Gramática __based](#)
- [Ponteiros com base](#)

Confira também

[Modificadores específicos da Microsoft](#)

Gramática __based

02/09/2020 • 2 minutes to read • [Edit Online](#)

Específico da Microsoft

O endereçamento baseado é útil quando é necessário controle preciso sobre o segmento ao qual os objetos estão alocados (dados estáticos e com base dinâmica).

A única forma de endereçamento com base aceitável em compilações de 32 bits e 64 bits é "baseada em um ponteiro" que define um tipo que contém um deslocamento de 32 bits ou de 64 bits para uma base de 32 bits ou de 64 bits ou com base em `void` .

Gramática

com base em intervalo-modificador. __based (expressão base)

expressão base. com base em variablebased-abstract-declaratorsegment-namesegment-Cast

variável com base. identificador

based-abstract-declarator. abstract-declarator

tipo base. tipo-nome

FINAL específico da Microsoft

Confira também

[Ponteiros baseados](#)

Ponteiros baseados (C++)

02/09/2020 • 2 minutes to read • [Edit Online](#)

A `__based` palavra-chave permite declarar ponteiros com base em ponteiros (ponteiros que são deslocamentos de ponteiros existentes). A `__based` palavra-chave é específica da Microsoft.

Sintaxe

```
type __based( base ) declarator
```

Comentários

Ponteiros baseados em endereços de ponteiro são a única forma da `__based` palavra-chave válida em compilações de 32 bits ou 64 bits. Para os compiladores C/C++ de 32 bits da Microsoft, um ponteiro baseado é um deslocamento de 32 bits de uma base de ponteiro de 32 bits. Uma restrição semelhante é mantida para ambientes de 64 bits, onde um ponteiro baseado é um deslocamento de 64 bits da base de 64 bits.

Um uso para ponteiros baseados em ponteiros é para identificadores persistentes que contêm ponteiros. Uma lista vinculada que consiste em ponteiros baseados em um ponteiro pode ser salva em disco e depois ser recarregada em outro local na memória, com os ponteiros permanecendo válidos. Por exemplo:

```
// based_pointers1.cpp
// compile with: /c
void *vpBuffer;
struct llist_t {
    void __based( vpBuffer ) *vpData;
    struct llist_t __based( vpBuffer ) *llNext;
};
```

O ponteiro `vpBuffer` é atribuído ao endereço da memória alocada em algum momento posterior no programa. A lista vinculada é realocada em relação ao valor de `vpBuffer`.

NOTE

Os identificadores de persistência contendo ponteiros também podem ser obtidos com o uso [de arquivos mapeados para memória](#).

Ao desreferenciar um ponteiro baseado, a base deve ser especificada explicitamente ou implicitamente conhecida na declaração.

Para compatibilidade com versões anteriores, `_based` é um sinônimo para `__based` a menos que a opção do compilador [/za \(desabilitar extensões de linguagem\)](#) seja especificada.

Exemplo

O código a seguir demonstra a alteração de um ponteiro baseado alterando sua base.

```
// based_pointers2.cpp
// compile with: /EHsc
#include <iostream>

int a1[] = { 1,2,3 };
int a2[] = { 10,11,12 };
int *pBased;

typedef int __based(pBased) * pBasedPtr;

using namespace std;
int main() {
    pBased = &a1[0];
    pBasedPtr pb = 0;

    cout << *pb << endl;
    cout << *(pb+1) << endl;

    pBased = &a2[0];

    cout << *pb << endl;
    cout << *(pb+1) << endl;
}
```

```
1
2
10
11
```

Confira também

[Palavras-chave](#)
[alloc_text](#)

Convenções de chamada

02/09/2020 • 2 minutes to read • [Edit Online](#)

O compilador Visual C/C++ fornece várias convenções diferentes para chamar funções internas e externas. Entender essas abordagens diferentes pode ajudar a depurar seu programa e a vincular seu código a rotinas de linguagem de assembly.

Os tópicos neste assunto explicam as diferenças entre as convenções de chamada, como os argumentos são passados, e como os valores são retornados por funções. Também são abordadas chamadas de função naked, um recurso avançado que permite escrever seu próprio código de prólogo e epílogo.

Para obter informações sobre como chamar convenções para processadores x64, consulte [Convenção de chamada](#).

Tópicos desta seção

- [Passagem de argumentos e convenções de nomenclatura](#) (`__cdecl` , `__stdcall` , `__fastcall` e outros)
- [Exemplo de chamada: protótipo de função e chamada](#)
- [Usando chamadas de função naked para escrever o código de prólogo/epílogo personalizado](#)
- [Coprocessador de ponto flutuante e convenções de chamada](#)
- [Convenções de chamada obsoletas](#)

Confira também

[Modificadores específicos da Microsoft](#)

Passagem de argumento e convenções de nomenclatura

02/09/2020 • 3 minutes to read • [Edit Online](#)

Específico da Microsoft

Os compiladores do Microsoft C++ permitem que você especifique convenções para passar argumentos e valores de retorno entre funções e chamadores. Nem todas as convenções estão disponíveis em todas as plataformas com suporte. Além disso, algumas convenções usam implementações específicas para a plataforma. Na maioria dos casos, as palavras-chave ou opções de compilador que especificam uma convenção sem suporte em uma plataforma específica são ignoradas e a convenção padrão de plataforma é usada.

Nas plataformas x86, todos os argumentos são ampliados para 32 bits quando passados. Os valores de retorno também são ampliados para 32 bits e retornados no registro de EAX, com exceção das estruturas de 8 bytes, que são retornadas no par de registro EDX:EAX. Estruturas maiores são retornadas no registro de EAX como ponteiros para estruturas de retorno ocultas. Os parâmetros são empurrados para a pilha da direita para a esquerda. As estruturas que não forem PODs não serão retornadas em registros.

O compilador gera códigos de prólogo e de epílogo para salvar e restaurar os registros de ESI, EDI, EBX e EBP, se eles forem usados na função.

NOTE

Quando um estrutura, união ou classe é retornada de uma função pelo valor, todas as definições de tipo precisam ser iguais, caso contrário, o programa poderá falhar no runtime.

Para obter informações sobre como definir seu próprio prólogo de função e código epílogo, consulte [chamadas de função Naked](#).

Para obter informações sobre as convenções de chamada padrão em código direcionado a plataformas x64, consulte [Convenção de chamada x64](#). Para obter informações sobre como chamar problemas de Convenção no código direcionado a plataformas ARM, consulte [problemas comuns de migração do Visual C++ ARM](#).

As seguintes convenções de chamada são suportadas pelo compilador visual do C/C++.

PALAVRA-CHAVE	LIMPEZA DA PILHA	PASSAGEM DE PARÂMETRO
<code>_cdecl</code>	Chamador	Empurra parâmetros para a pilha, em ordem inversa (direita para a esquerda)
<code>_clrcall</code>	N/D	Carrega parâmetros na pilha de expressões CLR em ordem (da esquerda para a direita).
<code>_stdcall</code>	Receptor	Empurra parâmetros para a pilha, em ordem inversa (direita para a esquerda)
<code>_fastcall</code>	Receptor	Armazenado em registros, em seguida, empurrado para a pilha

PALAVRA-CHAVE	LIMPEZA DA PILHA	PASSAGEM DE PARÂMETRO
__thiscall	Receptor	Enviado por push na pilha; <code>this</code> ponteiro armazenado no ecx
__vectorcall	Receptor	Armazenado em registros e depois empurrado na pilha na ordem inversa (da direita para a esquerda)

Para obter informações relacionadas, consulte [convenções de chamada obsoletas](#).

FINAL específico da Microsoft

Confira também

[Convenções de chamada](#)

__cdecl

02/09/2020 • 3 minutes to read • [Edit Online](#)

__cdecl é a Convenção de chamada padrão para programas C e C++. Como a pilha é limpa pelo chamador, ela pode executar vararg funções. A __cdecl Convenção de chamada cria executáveis maiores do que __stdcall, pois requer cada chamada de função para incluir o código de limpeza da pilha. A lista a seguir mostra a implementação dessa convenção de chamada. O __cdecl modificador é específico da Microsoft.

ELEMENTO	IMPLEMENTAÇÃO
Ordem de passagem de argumentos	Da direita para a esquerda.
Responsabilidade de manutenção de pilha	A função de chamada remove os argumentos da pilha.
Convenção de decoração de nome	O caractere de sublinhado (_) é prefixado para nomes, exceto quando __cdecl funções que usam vínculo de C são exportadas.
Convenção de conversão de maiúsculas/minúsculas	Nenhuma conversão de maiúsculas/minúsculas é realizada.

NOTE

Para obter informações relacionadas, consulte [nomes decorados](#).

Coloque o __cdecl modificador antes de uma variável ou um nome de função. Como as convenções de nomenclatura e chamada C são o padrão, a única vez que você deve usar __cdecl no código x86 é quando você especificou a /GV opção de compilador (vectorcall), /GZ (stdcall) ou /GR (fastcall). A opção de compilador /GD força a __cdecl Convenção de chamada.

Em processadores ARM e x64, __cdecl é aceito, mas normalmente ignorado pelo compilador. Por convenção no ARM e x64, os argumentos são passados nos registros quando possível, e os argumentos subsequentes são passados na pilha. No código x64, use __cdecl para substituir a opção de compilador /GV e use a Convenção de chamada x64 padrão.

Para funções de classe não estáticas, se a função for definida como fora da linha, o modificador da convenção de chamada não precisará ser especificado na definição fora da linha. Ou seja, para métodos de membro de classe não estática, a convenção de chamada especificada durante a declaração é assumida no ponto de definição. Dada esta definição de classe:

```
struct CMyClass {  
    void __cdecl mymethod();  
};
```

isto:

```
void CMyClass::mymethod() { return; }
```

equivale a isto:

```
void __cdecl CMyClass::mymethod() { return; }
```

Para compatibilidade com versões anteriores, `cdecl` e `_cdecl` são um sinônimo para `__cdecl`, a menos que a opção do compilador [/za \(desabilitar extensões de linguagem\)](#) seja especificada.

Exemplo

No exemplo a seguir, o compilador é instruído para usar convenções de nomenclatura e chamada em C para a função `system`.

```
// Example of the __cdecl keyword on function
int __cdecl system(const char *);
// Example of the __cdecl keyword on function pointer
typedef BOOL (_cdecl *funcname_ptr)(void * arg1, const char * arg2, DWORD flags, ...);
```

Confira também

[Passagem de argumentos e convenções de nomenclatura](#)

[Palavras-chave](#)

__clrcall

06/12/2019 • 5 minutes to read • [Edit Online](#)

Especifica que uma função só pode ser chamada de um código gerenciado. Use `__clrcall` para todas as funções virtuais que só serão chamadas a partir do código gerenciado. No entanto, essa convenção de chamada não pode ser usada para as funções que serão chamadas a partir do código nativo. O modificador de `__clrcall` é específico da Microsoft.

Use `__clrcall` para melhorar o desempenho ao chamar de uma função gerenciada para uma função gerenciada virtual ou da função gerenciada para a função gerenciada por meio de ponteiro.

Os pontos de entrada são funções separadas geradas pelo compilador. Se uma função tem pontos de entrada nativos e gerenciados, um deles será a função real com a implementação da função. Outra função será uma função separada (uma conversão) que chamará a função real e deixará Common Language Runtime executar PInvoke. Ao marcar uma função como `__clrcall`, você indica que a implementação da função deve ser MSIL e que a função de ponto de entrada nativa não será gerada.

Ao pegar o endereço de uma função nativa se `__clrcall` não for especificado, o compilador usará o ponto de entrada nativo. `__clrcall` indica que a função é gerenciada e não há necessidade de passar pela transição de gerenciado para nativo. Nesse caso, o compilador usa o ponto de entrada gerenciado.

Quando `/clr` (não `/clr:pure` ou `/clr:safe`) é usado e `__clrcall` não é usado, pegar o endereço de uma função sempre retorna o endereço da função de ponto de entrada nativa. Quando `__clrcall` é usado, a função de ponto de entrada nativa não é criada e, portanto, você obtém o endereço da função gerenciada, não uma função de conversão de ponto de entrada. Para obter mais informações, consulte [conversão dupla](#). As opções de compilador `/CLR: Pure` e `/CLR: safe` são preteridas no Visual Studio 2015 e sem suporte no Visual Studio 2017.

o [/CLR \(compilação Common Language Runtime\)](#) implica que todas as funções e ponteiros de função são `__clrcall` e o compilador não permitirá que uma função dentro do compilador seja marcada como algo diferente de `__clrcall`. Quando `/CLR: Pure` é usado, `__clrcall` só pode ser especificado em ponteiros de função e em declarações externas.

Você pode chamar diretamente `__clrcall` funções do código C++ existente que foi compilado usando `/CLR`, desde que essa função tenha uma implementação MSIL. `__clrcall` funções não podem ser chamadas diretamente de funções que têm asm embutido e chamam intrinsics específicas de CPU, por exemplo, mesmo que essas funções sejam compiladas com `/clr`.

`__clrcall` ponteiros de função devem ser usados apenas no domínio do aplicativo no qual foram criados. Em vez de passar `__clrcall` ponteiros de função entre domínios de aplicativo, use [CrossAppDomainDelegate](#). Para obter mais informações, consulte [domínios de aplicativo C++ e Visual](#).

Exemplo

Observe que quando uma função é declarada com `__clrcall`, o código será gerado quando necessário; por exemplo, quando a função é chamada.

```

// clrcall2.cpp
// compile with: /clr
using namespace System;
int __clrcall Func1() {
    Console::WriteLine("in Func1");
    return 0;
}

// Func1 hasn't been used at this point (code has not been generated),
// so runtime returns the address of a stub to the function
int (__clrcall *pf)() = &Func1;

// code calls the function, code generated at difference address
int i = pf(); // comment this line and comparison will pass

int main() {
    if (&Func1 == pf)
        Console::WriteLine("&Func1 == pf, comparison succeeds");
    else
        Console::WriteLine("&Func1 != pf, comparison fails");

    // even though comparison fails, stub and function call are correct
    pf();
    Func1();
}

```

```

in Func1
&Func1 != pf, comparison fails
in Func1
in Func1

```

Exemplo

O exemplo a seguir mostra que você pode definir um ponteiro de função de modo a declarar que o ponteiro de função será invocado apenas de códigos gerenciados. Isso permite que o compilador chame diretamente a função gerenciada e evita o ponto de entrada nativo (problema de duas conversões).

```

// clrcall3.cpp
// compile with: /clr
void Test() {
    System::Console::WriteLine("in Test");
}

int main() {
    void (*pTest)() = &Test;
    (*pTest)();

    void (__clrcall *pTest2)() = &Test;
    (*pTest2)();
}

```

Consulte também

[Convenções de passagem e nomenclatura de argumentos](#)

[Palavras-chave](#)

__stdcall

02/09/2020 • 3 minutes to read • [Edit Online](#)

A `__stdcall` Convenção de chamada é usada para chamar funções de API do Win32. O receptor limpa a pilha, portanto, o compilador faz `vararg` funções `__cdecl`. As funções que usam esta convenção de chamada exigem um protótipo de função. O `__stdcall` modificador é específico da Microsoft.

Sintaxe

```
tipo __stdcall de retorno Function-Name[ ( lista de argumentos ) ]
```

Comentários

A lista a seguir mostra a implementação dessa convenção de chamada.

ELEMENTO	IMPLEMENTAÇÃO
Ordem de passagem de argumentos	Da direita para a esquerda.
Convenção de passagem de argumentos	Por valor, a menos que um ponteiro ou um tipo de referência seja passado.
Responsabilidade de manutenção de pilha	A função chamada remove seus próprios argumentos da pilha.
Convenção de decoração de nome	Um sublinhado (<code>_</code>) é prefixado para o nome. O nome é seguido pelo sinal de arroba (<code>@</code>) seguido pelo número de bytes (em decimal) na lista de argumentos. Portanto, a função declarada como <code>int func(int a, double b)</code> está decorada da seguinte maneira: <code>_func@12</code>
Convenção de conversão de maiúsculas/minúsculas	Nenhum

A opção de compilador `/gz` especifica `__stdcall` para todas as funções não declaradas explicitamente com uma Convenção de chamada diferente.

Para compatibilidade com versões anteriores, `_stdcall` é um sinônimo para `__stdcall` a menos que a opção do compilador `/za` (desabilite extensões de linguagem) seja especificada.

As funções declaradas usando o `__stdcall` modificador retornam valores da mesma maneira que as funções declaradas usando `__cdecl`.

Em processadores ARM e x64, `__stdcall` é aceito e ignorado pelo compilador; em arquiteturas ARM e x64, por convenção, os argumentos são passados em registros quando possível, e os argumentos subsequentes são passados na pilha.

Para funções de classe não estáticas, se a função for definida como fora da linha, o modificador da convenção de chamada não precisará ser especificado na definição fora da linha. Ou seja, para métodos de membro de classe não estática, a convenção de chamada especificada durante a declaração é assumida no ponto de definição. Dada esta definição de classe,

```
struct CMyClass {  
    void __stdcall mymethod();  
};
```

this

```
void CMyClass::mymethod() { return; }
```

equivale a isso

```
void __stdcall CMyClass::mymethod() { return; }
```

Exemplo

No exemplo a seguir, o uso de `__stdcall` resultados em todos os `WINAPI` tipos de função tratados como uma chamada padrão:

```
// Example of the __stdcall keyword  
#define WINAPI __stdcall  
// Example of the __stdcall keyword on function pointer  
typedef BOOL (_stdcall *funcname_ptr)(void * arg1, const char * arg2, DWORD flags, ...);
```

Confira também

[Passagem de argumentos e convenções de nomenclatura](#)

[Palavras-chave](#)

__fastcall

02/09/2020 • 3 minutes to read • [Edit Online](#)

Específico da Microsoft

A `__fastcall` Convenção de chamada especifica que os argumentos para funções devem ser passados em registros, quando possível. Esta convenção de chamada se aplica somente à arquitetura x86. A lista a seguir mostra a implementação dessa convenção de chamada.

ELEMENTO	IMPLEMENTAÇÃO
Ordem de passagem de argumentos	Os dois primeiros argumentos de DWORD ou menores encontrados na lista de argumentos da esquerda para a direita são passados em registros de ECX e de EDX; todos os outros argumentos são passados na pilha da direita para a esquerda.
Responsabilidade de manutenção de pilha	A função de chamada retira os argumentos da pilha.
Convenção de decoração de nome	O sinal de arroba (@) é prefixado para nomes; um sinal de arroba seguido pelo número de bytes (em decimal) na lista de parâmetros tem o sufixo para nomes.
Convenção de conversão de maiúsculas/minúsculas	Nenhuma conversão de maiúsculas/minúsculas é realizada.

NOTE

As versões futuras do compilador podem usar registros diferentes para armazenar parâmetros.

Usar a opção de compilador `/gr` faz com que cada função no módulo seja compilada como a `__fastcall` menos que a função seja declarada usando um atributo conflitante ou o nome da função seja `main`.

A `__fastcall` palavra-chave é aceita e ignorada pelos compiladores que visam arquiteturas ARM e x64; em um chip x64, por convenção, os primeiros quatro argumentos são passados em registros quando possível, e argumentos adicionais são passados na pilha. Para obter mais informações, consulte [Convenção de chamada x64](#). Em um chip ARM, até quatro argumentos inteiros e oito argumentos de ponto flutuante podem ser passados em registros, e os argumentos adicionais são passados na pilha.

Para funções de classe não estáticas, se a função for definida como fora da linha, o modificador da convenção de chamada não precisará ser especificado na definição fora da linha. Ou seja, para métodos de membro de classe não estática, a convenção de chamada especificada durante a declaração é assumida no ponto de definição. Dada esta definição de classe:

```
struct CMyClass {  
    void __fastcall mymethod();  
};
```

isto:

```
void CMyClass::mymethod() { return; }
```

equivale a isto:

```
void __fastcall CMyClass::mymethod() { return; }
```

Para compatibilidade com versões anteriores, `_fastcall` é um sinônimo para `__fastcall` a menos que a opção do compilador [/za \(desabilitar extensões de linguagem\)](#) seja especificada.

Exemplo

No exemplo a seguir, a função `DeleteAggrWrapper` recebe argumentos passados em registros:

```
// Example of the __fastcall keyword
#define FASTCALL __fastcall

void FASTCALL DeleteAggrWrapper(void* pWrapper);
// Example of the __fastcall keyword on function pointer
typedef BOOL (__fastcall *funcname_ptr)(void * arg1, const char * arg2, DWORD flags, ...);
```

FINAL específico da Microsoft

Confira também

[Passagem de argumentos e convenções de nomenclatura](#)

[Palavras-chave](#)

A Convenção de chamada **específica da Microsoft** `__thiscall` é usada em funções de membro de classe C++ na arquitetura x86. É a Convenção de chamada padrão usada pelas funções de membro que não usam argumentos variáveis (`vararg` Functions).

Em `__thiscall` , o receptor limpa a pilha, o que é impossível para as `vararg` funções. Os argumentos são enviados por push na pilha da direita para a esquerda. O `this` ponteiro é transmitido por meio do Registro ECX e não da pilha.

Em máquinas ARM, ARM64 e x64, `__thiscall` é aceita e ignorada pelo compilador. Isso ocorre porque eles usam uma Convenção de chamada baseada em registro por padrão.

Um motivo para usar `__thiscall` está em classes cujas funções de membro usam `__clrcall` por padrão. Nesse caso, você pode usar `__thiscall` para tornar as funções de membro individuais que podem ser chamadas do código nativo.

Ao compilar com `/clr:pure` , todas as funções e ponteiros de função são `__clrcall` , a menos que especificado o contrário. As `/clr:pure` `/clr:safe` Opções de compilador e são preteridas no visual Studio 2015 e sem suporte no visual Studio 2017.

`vararg` as funções de membro usam a `__cdecl` Convenção de chamada. Todos os argumentos de função são enviados por push na pilha, com o `this` ponteiro colocado na pilha por último.

Como essa Convenção de chamada se aplica apenas ao C++ , ela não tem um esquema de decoração de nome de C.

Quando você define uma função de membro de classe não estática fora de linha, especifique o modificador de Convenção de chamada somente na declaração. Você não precisa especificá-lo novamente na definição fora de linha. O compilador usa a Convenção de chamada especificada durante a declaração no ponto de definição.

Consulte também

[Passagem de argumento e convenções de nomenclatura](#)

__vectorcall

02/09/2020 • 21 minutes to read • [Edit Online](#)

Específico da Microsoft

A `__vectorcall` Convenção de chamada especifica que os argumentos para funções devem ser passados em registros, quando possível. `__vectorcall` usa mais registros para argumentos do que `__fastcall` ou o uso padrão de [Convenção de chamada x64](#). A `__vectorcall` Convenção de chamada só tem suporte em código nativo em processadores x86 e x64 que incluem Streaming SIMD Extensions 2 (SSE2) e superior. Use `__vectorcall` para acelerar as funções que passam vários argumentos de vetor de ponto flutuante ou SIMD e executam operações que aproveitam os argumentos carregados nos registros. A lista a seguir mostra os recursos que são comuns às implementações x86 e x64 do `__vectorcall`. As diferenças são explicadas posteriormente neste artigo.

ELEMENTO	IMPLEMENTAÇÃO
Convenção de nomes decoração do C	Os nomes de função têm um sufixo com dois sinais "at" (@ @) seguidos pelo número de bytes (em decimal) na lista de parâmetros.
Convenção de conversão de maiúsculas/minúsculas	Nenhuma tradução realizada.

O uso da `/Gv` opção do compilador faz com que cada função no módulo compile como a `__vectorcall` menos que a função seja uma função membro, seja declarada com um atributo de Convenção de chamada conflitante, use uma `vararg` lista de argumentos variáveis ou tenha o nome `main`.

Você pode passar três tipos de argumentos pelo registro em `__vectorcall` funções: valores de *tipo de inteiro*, valores de *tipo de vetor* e valores de HVA (*agregação de vetor homogêneo*).

Um tipo inteiro satisfaz dois requisitos: se encaixa no tamanho nativo de registro do processador, por exemplo, 4 bytes em um computador x86 ou 8 bytes em um computador x64, e pode ser convertido para um número inteiro do comprimento de registro e de volta para o seu formato original sem alterar sua representação de bit. Por exemplo, qualquer tipo que possa ser promovido para `int` no x86 (`long long` em x64) — por exemplo, um `char` ou `short` — ou que possa ser convertido em `int` (`long long` em x64) e de volta para seu tipo original sem alteração é um tipo inteiro. Tipos de inteiros incluem ponteiro, referência e `struct` ou `union` tipos de 4 bytes (8 bytes em x64) ou menos. Em plataformas x64, maiores `struct` e `union` tipos são passados por referência à memória alocada pelo chamador; em plataformas x86, eles são passados por valor na pilha.

Um tipo de vetor é um tipo de ponto flutuante — por exemplo, a `float` ou `double` — ou um tipo de vetor SIMD — por exemplo, `__m128` ou `__m256`.

Um tipo de HVA é um tipo composto de até quatro membros de dados que têm tipos vetoriais idênticos. Um tipo de HVA tem o mesmo requisito de alinhamento que o tipo de vetor dos seus membros. Este é um exemplo de uma definição de HVA `struct` que contém três tipos de vetor idênticos e tem um alinhamento de 32 bytes:

```
typedef struct {
    __m256 x;
    __m256 y;
    __m256 z;
} hva3;    // 3 element HVA type on __m256
```

Declare suas funções explicitamente com a `__vectorcall` palavra-chave em arquivos de cabeçalho para permitir que o código compilado separadamente se vincule sem erros. As funções devem ter um protótipo para usar `__vectorcall` e não podem usar uma `vararg` lista de argumentos de comprimento variável.

Uma função de membro pode ser declarada usando o `__vectorcall` especificador. O `this` ponteiro oculto é passado pelo registro como o primeiro argumento de tipo inteiro.

Em máquinas ARM, `__vectorcall` é aceita e ignorada pelo compilador.

Para funções de membro de classe não estáticas, se a função for definida como fora da linha, o modificador da convenção de chamada não precisa ser especificado na definição fora da linha. Ou seja, para membros de classe não estática, a convenção de chamada especificada durante a declaração é assumida no ponde de definição. Dada esta definição de classe:

```
struct MyClass {  
    void __vectorcall mymethod();  
};
```

isto:

```
void MyClass::mymethod() { return; }
```

equivale a isto:

```
void __vectorcall MyClass::mymethod() { return; }
```

O `__vectorcall` modificador de Convenção de chamada deve ser especificado quando um ponteiro para uma `__vectorcall` função é criado. O exemplo a seguir cria um `typedef` para um ponteiro para uma `__vectorcall` função que usa quatro `double` argumentos e retorna um `__m256` valor:

```
typedef __m256 (__vectorcall * vcfnptr)(double, double, double, double);
```

Para compatibilidade com versões anteriores, `__vectorcall` é um sinônimo para `__vectorcall1` a menos que a opção do compilador `/Za` (desabilite extensões de linguagem) seja especificada.

convenção de `__vectorcall` em x64

A `__vectorcall1` Convenção de chamada em x64 estende a Convenção de chamada Standard x64 para aproveitar os registros adicionais. Os argumentos de tipo inteiro e os argumentos de tipo vetor são mapeados nos registradores com base em sua posição na lista de argumentos. Os argumentos HVA são alocados a registros de vetor não utilizados.

Quando alguns dos primeiros quatro argumentos na ordem são argumentos de tipo inteiro da esquerda para a direita, eles são passados no registro correspondente a essa posição - RCX, RDX, R8, ou R9. Um `this` ponteiro oculto é tratado como o primeiro argumento de tipo inteiro. Quando um argumento de HVA em um dos primeiros quatro argumentos não pode ser passado em registros disponíveis, uma referência à memória atribuída ao chamador é passada no registro correspondente do tipo inteiro em vez disso. Os argumentos de tipo inteiro após a quarta posição de parâmetro são passados para a pilha.

Quando algum dos primeiros seis argumentos na ordem são argumentos de tipo vetor da esquerda para a direita, eles são passados por valor nos registros de vetor SSE 0 a 5, de acordo com a posição do argumento. Os tipos e ponto flutuante `__m128` são passados em registros XMM e os `__m256` tipos são passados em registros YMM. Isso difere da convenção padrão de chamada do x64, porque os tipos de vetor são passados por valor, não por

referência, e registros adicionais são usados. O espaço de pilha de sombra alocado para argumentos de tipo de vetor é fixo em 8 bytes e a `/homeparams` opção não se aplica. Os argumentos de tipo vetorial na posição de parâmetro sete ou posteriores são passados na pilha em função de memória atribuída pelo chamador.

Depois que os registros são alocados para argumentos de vetor, os membros de dados dos argumentos HVA são alocados, em ordem crescente, para o vetor não usado registra XMM0 em XMM5 (ou YMM0 para YMM5, para `_m256` tipos), desde que haja registros suficientes disponíveis para o HVA inteiro. Se não houver registros disponíveis, o argumento do HVA será passado por referência para a memória alocada pelo chamador. O espaço de sombra de pilha para um argumento de HVA é corrigido em 8 bytes com conteúdo indefinido. Os argumentos HVA são atribuídos a registros da esquerda para a direita na lista de parâmetros e podem estar em qualquer posição. Os argumentos HVA em uma das primeiras quatro posições de argumento que não são atribuídos a registros de vetor são passados por referência no registro de inteiro que corresponda a essa posição. Os argumentos HVA passados por referência após a quarta posição de parâmetro são enviados por push na pilha.

Os resultados das `_vectorcall` funções são retornados por valor em registros quando possível. Os resultados do tipo inteiro, incluindo as estruturas ou uniões de 8 bytes ou menos, são retornadas pelo valor em RAX. Os resultados do tipo vetorial são retornados pelo valor em XMM0 ou em YMM0, dependendo do tamanho. Os resultados HVA têm cada elemento de dados retornado pelo valor em registros XMM0:XMM3 ou YMM0:YMM3, dependendo do tamanho do elemento. Os tipos de resultado que não cabem nos registros correspondentes são retornados por referência à memória alocada pelo chamador.

A pilha é mantida pelo chamador na implementação x64 do `_vectorcall`. O código de prólogo e epílogo do chamador aloca e limpa a pilha da função chamada. Os argumentos são empurrados na pilha da direita para a esquerda, e o espaço da pilha de sombra é atribuído para os argumentos passados nos registros.

Exemplos:

```
// crt_vc64.c
// Build for amd64 with: cl /arch:AVX /W3 /FAs crt_vc64.c
// This example creates an annotated assembly listing in
// crt_vc64.asm.

#include <intrin.h>
#include <xmmmintrin.h>

typedef struct {
    __m128 array[2];
} hva2;      // 2 element HVA type on __m128

typedef struct {
    __m256 array[4];
} hva4;      // 4 element HVA type on __m256

// Example 1: All vectors
// Passes a in XMM0, b in XMM1, c in YMM2, d in XMM3, e in YMM4.
// Return value in XMM0.
__m128 __vectorcall
example1(__m128 a, __m128 b, __m256 c, __m128 d, __m256 e) {
    return d;
}

// Example 2: Mixed int, float and vector parameters
// Passes a in RCX, b in XMM1, c in R8, d in XMM3, e in YMM4,
// f in XMM5, g pushed on stack.
// Return value in YMM0.
__m256 __vectorcall
example2(int a, __m128 b, int c, __m128 d, __m256 e, float f, int g) {
    return e;
}

// Example 3: Mixed int and HVA parameters
// Passes a in RCX, c in R8, d in R9, and e pushed on stack.
```

```

// Passes b by element in [XMM0:XMM1];
// b's stack shadow area is 8-bytes of undefined value.
// Return value in XMM0.
__m128 __vectorcall example3(int a, hva2 b, int c, int d, int e) {
    return b.array[0];
}

// Example 4: Discontiguous HVA
// Passes a in RCX, b in XMM1, d in XMM3, and e is pushed on stack.
// Passes c by element in [YMM0,YMM2,YMM4,YMM5], discontiguous because
// vector arguments b and d were allocated first.
// Shadow area for c is an 8-byte undefined value.
// Return value in XMM0.
float __vectorcall example4(int a, float b, hva4 c, __m128 d, int e) {
    return b;
}

// Example 5: Multiple HVA arguments
// Passes a in RCX, c in R8, e pushed on stack.
// Passes b in [XMM0:XMM1], d in [YMM2:YMM5], each with
// stack shadow areas of an 8-byte undefined value.
// Return value in RAX.
int __vectorcall example5(int a, hva2 b, int c, hva4 d, int e) {
    return c + e;
}

// Example 6: HVA argument passed by reference, returned by register
// Passes a in [XMM0:XMM1], b passed by reference in RDX, c in YMM2,
// d in [XMM3:XMM4].
// Register space was insufficient for b, but not for d.
// Return value in [YMM0:YMM3].
hva4 __vectorcall example6(hva2 a, hva4 b, __m256 c, hva2 d) {
    return b;
}

int __cdecl main( void )
{
    hva4 h4;
    hva2 h2;
    int i;
    float f;
    __m128 a, b, d;
    __m256 c, e;

    a = b = d = _mm_set1_ps(3.0f);
    c = e = _mm256_set1_ps(5.0f);
    h2.array[0] = _mm_set1_ps(6.0f);
    h4.array[0] = _mm256_set1_ps(7.0f);

    b = example1(a, b, c, d, e);
    e = example2(1, b, 3, d, e, 6.0f, 7);
    d = example3(1, h2, 3, 4, 5);
    f = example4(1, 2.0f, h4, d, 5);
    i = example5(1, h2, 3, h4, 5);
    h4 = example6(h2, h4, c, h2);
}

```

convenção de __vectorcall em x86

A `__vectorcall` Convenção de chamada segue a `__fastcall` Convenção para argumentos de tipo inteiro de 32 bits e aproveita os registros de vetor SSE para os argumentos tipo de vetor e HVA.

Os dois primeiros argumentos de tipo inteiro localizados na lista de parâmetros, da esquerda para a direita, são colocados em ECX e EDX, respectivamente. Um `this` ponteiro oculto é tratado como o primeiro argumento de tipo inteiro e é passado em ecx. Os primeiros seis argumentos de tipo de vetor são transmitidos por valor pelos

registros 0 a 5 do vetor SSE, nos registros de MMX ou YMM, dependendo do tamanho do argumento.

Os primeiros seis argumentos de tipo de vetor, da esquerda para a direita, são transmitidos por valor nos registros 0 a 5 do vetor SSE. Os tipos e ponto flutuante `__m128` são passados em registros XMM e os `__m256` tipos são passados em registros YMM. Nenhum espaço de pilha de sombra está alocado para argumentos de tipo de vetor passado pelo registro. Os argumentos de tipo vetorial sete ou posteriores são passados na pilha em referência à memória atribuída pelo chamador. A limitação do erro do compilador [C2719](#) não se aplica a esses argumentos.

Depois que os registros são alocados para argumentos de vetor, os membros de dados dos argumentos HVA são alocados em ordem crescente para o vetor não utilizado regista XMM0 em XMM5 (ou YMM0 para YMM5, para `__m256` tipos), desde que haja registros suficientes disponíveis para o HVA inteiro. Se não houver registros disponíveis, o argumento do HVA será passado na pilha por referência para a memória alocada pelo chamador. Nenhum espaço de sombra de pilha para um argumento do HVA está alocado. Os argumentos HVA são atribuídos a registros da esquerda para a direita na lista de parâmetros e podem estar em qualquer posição.

Os resultados das `__vectorcall` funções são retornados por valor em registros quando possível. Os resultados do tipo inteiro, incluindo as estruturas ou uniões de 4 bytes ou menos, são retornadas pelo valor em EAX. As estruturas ou uniões de tipo inteiro de 8 bytes ou menos são retornadas por valor em EDX:EAX. Os resultados do tipo vetorial são retornados pelo valor em XMM0 ou em YMM0, dependendo do tamanho. Os resultados HVA têm cada elemento de dados retornado pelo valor em registros XMM0:XMM3 ou YMM0:YMM3, dependendo do tamanho do elemento. Outros tipos de resultados são retornados em referência à memória atribuída pelo chamador.

A implementação x86 do `__vectorcall` segue a Convenção de argumentos enviados por push na pilha da direita para a esquerda pelo chamador, e a função chamada limpa a pilha logo antes de retornar. Apenas os argumentos que não são colocados em registros são empurrados na pilha.

Exemplos:

```
// crt_vc86.c
// Build for x86 with: cl /arch:AVX /W3 /FAs crt_vc86.c
// This example creates an annotated assembly listing in
// crt_vc86.asm.

#include <intrin.h>
#include <xmmmintrin.h>

typedef struct {
    __m128 array[2];
} hva2;    // 2 element HVA type on __m128

typedef struct {
    __m256 array[4];
} hva4;    // 4 element HVA type on __m256

// Example 1: All vectors
// Passes a in XMM0, b in XMM1, c in YMM2, d in XMM3, e in YMM4.
// Return value in XMM0.
__m128 __vectorcall
example1(__m128 a, __m128 b, __m256 c, __m128 d, __m256 e) {
    return d;
}

// Example 2: Mixed int, float and vector parameters
// Passes a in ECX, b in XMM0, c in EDX, d in XMM1, e in YMM2,
// f in XMM3, g pushed on stack.
// Return value in YMM0.
__m256 __vectorcall
example2(int a, __m128 b, int c, __m128 d, __m256 e, float f, int g) {
    return e;
}
```

```

// Example 3: Mixed int and HVA parameters
// Passes a in ECX, c in EDX, d and e pushed on stack.
// Passes b by element in [XMM0:XMM1].
// Return value in XMM0.
__m128 __vectorcall example3(int a, hva2 b, int c, int d, int e) {
    return b.array[0];
}

// Example 4: HVA assigned after vector types
// Passes a in ECX, b in XMM0, d in XMM1, and e in EDX.
// Passes c by element in [YMM2:YMM5].
// Return value in XMM0.
float __vectorcall example4(int a, float b, hva4 c, __m128 d, int e) {
    return b;
}

// Example 5: Multiple HVA arguments
// Passes a in ECX, c in EDX, e pushed on stack.
// Passes b in [XMM0:XMM1], d in [YMM2:YMM5].
// Return value in EAX.
int __vectorcall example5(int a, hva2 b, int c, hva4 d, int e) {
    return c + e;
}

// Example 6: HVA argument passed by reference, returned by register
// Passes a in [XMM1:XMM2], b passed by reference in ECX, c in YMM0,
// d in [XMM3:XMM4].
// Register space was insufficient for b, but not for d.
// Return value in [YMM0:YMM3].
hva4 __vectorcall example6(hva2 a, hva4 b, __m256 c, hva2 d) {
    return b;
}

int __cdecl main( void )
{
    hva4 h4;
    hva2 h2;
    int i;
    float f;
    __m128 a, b, d;
    __m256 c, e;

    a = b = d = _mm_set1_ps(3.0f);
    c = e = _mm256_set1_ps(5.0f);
    h2.array[0] = _mm_set1_ps(6.0f);
    h4.array[0] = _mm256_set1_ps(7.0f);

    b = example1(a, b, c, d, e);
    e = example2(1, b, 3, d, e, 6.0f, 7);
    d = example3(1, h2, 3, 4, 5);
    f = example4(1, 2.0f, h4, d, 5);
    i = example5(1, h2, 3, h4, 5);
    h4 = example6(h2, h4, c, h2);
}

```

Fim da seção específica da Microsoft

Confira também

[Passagem de argumentos e convenções de nomenclatura](#)

[Palavras-chave](#)

Exemplo de chamada: protótipo de função e chamada

25/03/2020 • 2 minutes to read • [Edit Online](#)

Seção específica da Microsoft

O exemplo a seguir mostra os resultados de fazer uma chamada de função usando várias convenções de chamada.

Este exemplo é baseado no seguinte esqueleto da função. Substituir `calltype` com a convenção de chamada apropriada.

```
void    calltype MyFunc( char c, short s, int i, double f );
.

.

.

void    MyFunc( char c, short s, int i, double f )
{
.

.

.

}

.

.

.

MyFunc ('x', 12, 8192, 2.7183);
```

Para obter mais informações, consulte [resultados do exemplo de chamada](#).

Fim da seção específica da Microsoft

Confira também

[Convenções de chamada](#)

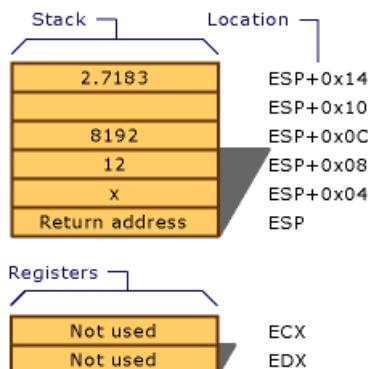
Resultados do exemplo de chamada

02/09/2020 • 2 minutes to read • [Edit Online](#)

Específico da Microsoft

cdecl

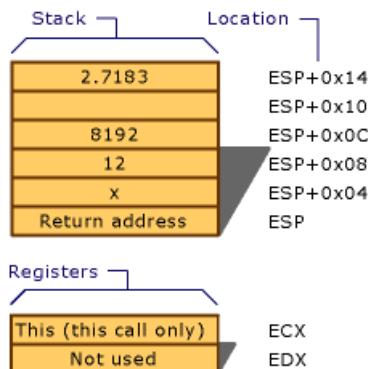
O nome da função decorada C é `_MyFunc` .



A `cdecl` Convenção de chamada

stdcall e thiscall

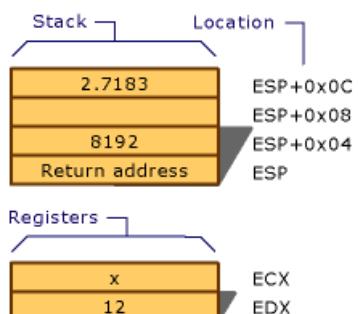
O nome decorado C (`__stdcall`) é `_MyFunc@20` . O nome decorado em C++ é específico da implementação.



As convenções de chamada `stdcall` e `thiscall`

fastcall

O nome decorado C (`__fastcall`) é `@MyFunc@20` . O nome decorado em C++ é específico da implementação.



A convenção de chamada `fastcall`

FINAL específico da Microsoft

Confira também

[Exemplo de chamada: protótipo de função e chamada](#)

Chamadas de função naked

02/09/2020 • 2 minutes to read • [Edit Online](#)

Específico da Microsoft

Funções declaradas com o `naked` atributo são emitidas sem o prólogo ou código epílogo, permitindo que você escreva suas próprias sequências de prólogo/epílogo personalizadas usando o [montador embutido](#). As funções naked são fornecidas como um recurso avançado. Elas permitem declarar uma função que está sendo chamada de um contexto diferente de C/C++ e, portanto, fazer suposições diferentes sobre onde estão os parâmetros ou quais registros são preservados. Os exemplos incluem rotinas como manipuladores de interrupção. Esse recurso é particularmente útil para gravadores de drivers de dispositivos virtuais (VxDs).

Que mais você deseja saber?

- [naked](#)
- [Regras e limitações para funções Naked](#)
- [Considerações para escrever código prólogo/epílogo](#)

FINAL específico da Microsoft

Confira também

[Convenções de chamada](#)

Regras e limitações para funções naked

02/09/2020 • 2 minutes to read • [Edit Online](#)

Específico da Microsoft

As seguintes regras e restrições se aplicam às funções naked:

- A `return` instrução não é permitida.
- O tratamento de exceções estruturado e tratamento de exceções de C++ não são permitidos, pois eles devem desenrolar pelo quadro de pilhas.
- Pela mesma razão, qualquer forma de `setjmp` é proibida.
- O uso da função `_alloca` é proibido.
- Para assegurar que nenhum código de inicialização para variáveis locais apareça antes da sequência de prólogo, variáveis locais inicializadas não são permitidas no escopo da função. Em particular, a declaração de objetos C++ não é permitida no escopo da função. Porém, podem haver dados inicializados em um escopo aninhado.
- A otimização do ponteiro do quadro (a opção /Oy do compilador) não é recomendada, mas ela será suprimida automaticamente para uma função naked.
- Você não pode declarar objetos da classe do C++ no escopo lexical da função. No entanto, é possível declarar objetos em um bloco aninhado.
- A `naked` palavra-chave é ignorada durante a compilação com [/CLR](#).
- Para `__fastcall` funções Naked, sempre que houver uma referência no código C/C++ para um dos argumentos de registro, o código de prólogo deverá armazenar os valores desse registro no local da pilha para essa variável. Por exemplo:

```
// nkdfastcl.cpp
// compile with: /c
// processor: x86
__declspec(naked) int __fastcall power(int i, int j) {
    // calculates i^j, assumes that j >= 0

    // prolog
    __asm {
        push ebp
        mov ebp, esp
        sub esp, __LOCAL_SIZE
        // store ECX and EDX into stack locations allocated for i and j
        mov i, ecx
        mov j, edx
    }

    {
        int k = 1;    // return value
        while (j-- > 0)
            k *= i;
        __asm {
            mov eax, k
        };
    }

    // epilog
    __asm {
        mov esp, ebp
        pop ebp
        ret
    }
}
```

FINAL específico da Microsoft

Confira também

[Chamadas de função Naked](#)

Considerações para escrever o código de prólogo/epílogo

15/04/2020 • 3 minutes to read • [Edit Online](#)

Específico da Microsoft

Antes de escrever suas próprias seqüências de código de prólogo e epílog, é importante entender como o quadro de pilha é definido. Também é útil saber como `__LOCAL_SIZE` usar o símbolo.

Layout do quadro de pilha

Este exemplo mostra o código padrão do prólogo que pode aparecer em uma função de 32 bits:

```
push    ebp          ; Save ebp
mov     ebp, esp      ; Set stack frame pointer
sub     esp, localbytes ; Allocate space for locals
push    <registers>   ; Save registers
```

A variável `localbytes` representa o número de bytes necessários na pilha para as variáveis locais. A variável `<registers>` é um espaço reservado que representa a lista de registros a serem salvos na pilha. Depois de enviar os registros, você pode colocar todos os outros dados apropriados na pilha. O seguinte exemplo é o código do epílogo correspondente:

```
pop    <registers>   ; Restore registers
mov     esp, ebp      ; Restore stack pointer
pop     ebp          ; Restore ebp
ret
```

A pilha sempre vai para baixo (dos endereços de memória mais altos para os mais baixos). O ponteiro de base (`ebp`) aponta para o valor enviado por push de `ebp`. A área de locais começa em `ebp-4`. Para acessar variáveis locais, calcule um deslocamento de `ebp` subtraindo o valor apropriado de `ebp`.

`__LOCAL_SIZE`

O compilador fornece um `__LOCAL_SIZE` símbolo, para uso no bloco de montagem inline do código de prólog de função. Esse símbolo é usado para alocar espaço para as variáveis locais no quadro da pilha no código personalizado de prólogo.

O compilador determina o `__LOCAL_SIZE` valor de . Seu valor é o número total de bytes de todas as variáveis locais definidas pelo usuário e variáveis temporárias geradas pelo compilador. `__LOCAL_SIZE` pode ser usado apenas como um operador imediato; não pode ser usado em uma expressão. Você não deve alterar ou redefinir o valor desse símbolo. Por exemplo:

```
mov     eax, __LOCAL_SIZE      ;Immediate operand--Okay
mov     eax, [ebp - __LOCAL_SIZE] ;Error
```

O exemplo a seguir de uma função nua contendo seqüências personalizadas de prólogo e epílog usa o `__LOCAL_SIZE` símbolo na seqüência de prólogs:

```
// the_local_size_symbol.cpp
// processor: x86
__declspec ( naked ) int main() {
    int i;
    int j;

    __asm {      /* prolog */
        push    ebp
        mov     ebp, esp
        sub     esp, __LOCAL_SIZE
    }

    /* Function body */
    __asm {      /* epilog */
        mov     esp, ebp
        pop    ebp
        ret
    }
}
```

Fim específico da Microsoft

Confira também

[Chamadas de função naked](#)

Coprocessador de ponto flutuante e convenções de chamada

02/09/2020 • 2 minutes to read • [Edit Online](#)

Se você estiver escrevendo rotinas de assembly para o coprocessador de ponto flutuante, você deve preservar a palavra de controle de ponto flutuante e limpar a pilha do coprocessador, a menos que você esteja retornando um `float` `double` valor ou (que sua função deve retornar em St (0)).

Confira também

[Convenções de chamada](#)

Convenções de chamada obsoletas

25/03/2020 • 2 minutes to read • [Edit Online](#)

Seção específica da Microsoft

Não há mais suporte para as convenções de chamada `__pascal`, `__fortran` e `__syscall`. Você pode emular a funcionalidade delas usando uma das convenções de chamada com suporte e as opções do vinculador apropriadas.

`<o Windows. h >` Agora dá suporte à macro `Winapi` que se traduz na Convenção de chamada apropriada para o destino. Use `Winapi` em vez de `PASCAL` ou `__far __pascal`.

Fim da seção específica da Microsoft

Confira também

[Convenções de passagem e nomenclatura de argumentos](#)

restrita (C++ AMP)

02/09/2020 • 3 minutes to read • [Edit Online](#)

O especificador de restrição pode ser aplicado a declarações de função e lambda. Impõe restrições no código na função e no comportamento da função em aplicativos que usam o runtime do C++ Accelerated Massive Parallelism(AMP C++).

NOTE

Para obter informações sobre a `restrict` palavra-chave que faz parte dos `__declspec` atributos de classe de armazenamento, consulte [restringir](#).

A `restrict` cláusula usa os seguintes formulários:

CLÁUSULA	DESCRIÇÃO
<code>restrict(cpu)</code>	A função pode usar a linguagem C++ completa. Somente outras funções que são declaradas usando as funções <code>restrict(cpu)</code> podem chamar a função.
<code>restrict(amp)</code>	A função só pode usar o subconjunto da linguagem C++ que o AMP C++ pode acelerar.
Uma sequência de <code>restrict(cpu)</code> e <code>restrict(amp)</code> .	A função deve atender às limitações de <code>restrict(cpu)</code> e <code>restrict(amp)</code> . A função pode ser chamada por funções que são declaradas usando <code>restrict(cpu)</code> , <code>restrict(amp)</code> , <code>restrict(cpu, amp)</code> ou <code>restrict(amp, cpu)</code> . A forma <code>restrict(A) restrict(B)</code> pode ser escrita como <code>restrict(A,B)</code> .

Comentários

A `restrict` palavra-chave é uma palavra-chave contextual. Os especificadores de restrição `cpu` e `amp` não são palavras reservadas. A lista de especificadores não é extensível. Uma função que não tem uma `restrict` cláusula é igual a uma função que tem a `restrict(cpu)` cláusula.

Uma função que tem a cláusula `restrict(amp)` tem as seguintes limitações:

- A função só pode chamar funções que tenham a cláusula `restrict(amp)`.
- A função deve ser embutível.
- A função pode declarar somente `int` variáveis, `unsigned int`, `float`, e `double`, e classes e estruturas que contêm apenas esses tipos. `bool` também é permitido, mas deve ser alinhado em 4 bytes se você usá-lo em um tipo composto.
- As funções lambda não podem capturar por referência nem capturar os ponteiros.
- As referências e ponteiros de indireção única só têm suporte como variáveis locais, argumentos de função e tipos de retorno.

- Os seguintes não são permitidos:
 - Recursão.
 - Variáveis declaradas com a palavra-chave `volatile`.
 - Funções virtuais.
 - Ponteiros para funções.
 - Ponteiros para funções de membro.
 - Ponteiros em estruturas.
 - Ponteiros para ponteiros.
 - `goto` instruções.
 - Instruções rotuladas.
 - `try **** catch` instruções, ou `throw`.
 - Variáveis globais.
 - Variáveis estáticas. Em vez disso, use [Tile_static](#) palavra-chave.
 - `dynamic_cast` conversões.
 - O `typeid` operador.
 - Declarações asm.
 - Varargs.

Para obter uma discussão sobre limitações de função, consulte [restrições \(amp\)](#).

Exemplo

O exemplo a seguir mostra como usar a `restrict(amp)` cláusula.

```
void functionAmp() restrict(amp) {}
void functionNonAmp() {}

void callFunctions() restrict(amp)
{
    // int is allowed.
    int x;
    // long long int is not allowed in an amp-restricted function. This generates a compiler error.
    // long long int y;

    // Calling an amp-restricted function is allowed.
    functionAmp();

    // Calling a non-amp-restricted function is not allowed.
    // functionNonAmp();
}
```

Confira também

[C++ AMP \(C++ Accelerated Massive Parallelism\)](#)

Palavra-chave tile_static

25/03/2020 • 4 minutes to read • [Edit Online](#)

A palavra-chave `tile_static` é usada para declarar uma variável que pode ser acessada por todos os threads em um bloco de threads. O tempo de vida da variável começa quando a execução alcança o ponto de declaração e termina com o retorno da função de kernel. Para obter mais informações sobre como usar blocos, consulte [usando blocos](#).

A palavra-chave `tile_static` tem as seguintes limitações:

- Pode ser usado somente em variáveis que estão em uma função que tenha o modificador `restrict(amp)`.
- Não pode ser usado em variáveis que são tipos de referência ou ponteiro.
- Uma variável `tile_static` não pode ter um inicializador. Os construtores e destruidores padrão não são invocados automaticamente.
- O valor de uma variável de `tile_static` não inicializada é indefinido.
- Se uma variável de `tile_static` for declarada em um grafo de chamada com raiz por uma chamada que não seja de ladrilho para `parallel_for_each`, um aviso será gerado e o comportamento da variável será indefinido.

Exemplo

O exemplo a seguir mostra como uma variável `tile_static` pode ser usada para acumular dados em vários threads em um bloco.

```
// Sample data:
int sampledata[] = {
    2, 2, 9, 7, 1, 4,
    4, 4, 8, 8, 3, 4,
    1, 5, 1, 2, 5, 2,
    6, 8, 3, 2, 7, 2};

// The tiles:
// 2 2      9 7      1 4
// 4 4      8 8      3 4
//
// 1 5      1 2      5 2
// 6 8      3 2      7 2

// Averages:
int averagedata[] = {
    0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0,
};

array_view<int, 2> sample(4, 6, sampledata);
array_view<int, 2> average(4, 6, averagedata);

parallel_for_each(
    // Create threads for sample.extent and divide the extent into 2 x 2 tiles.
    sample.extent.tile<2,2>(),
    [=](tiled_index<2,2> idx) restrict(amp)
    {
        // Create a 2 x 2 array to hold the values in this tile
        tile_static array<int, 2> tile_data;
        tile_data[0][0] = sample(idx[0], idx[1]);
        tile_data[0][1] = sample(idx[0], idx[1] + 1);
        tile_data[1][0] = sample(idx[0] + 1, idx[1]);
        tile_data[1][1] = sample(idx[0] + 1, idx[1] + 1);
        average[idx[0], idx[1]] = tile_data[0][0] + tile_data[0][1] + tile_data[1][0] + tile_data[1][1];
    }
);
```

```

// Create a 2 x 2 array to hold the values in this tile.
tile_static int nums[2][2];
// Copy the values for the tile into the 2 x 2 array.
nums[idx.local[1]][idx.local[0]] = sample[idx.global];
// When all the threads have executed and the 2 x 2 array is complete, find the average.
idx.barrier.wait();
int sum = nums[0][0] + nums[0][1] + nums[1][0] + nums[1][1];
// Copy the average into the array_view.
average[idx.global] = sum / 4;
}

};

for (int i = 0; i < 4; i++) {
    for (int j = 0; j < 6; j++) {
        std::cout << average(i,j) << " ";
    }
    std::cout << "\n";
}

// Output:
// 3 3 8 8 3 3
// 3 3 8 8 3 3
// 5 5 2 2 4 4
// 5 5 2 2 4 4
// Sample data.
int sampledata[] = {
    2, 2, 9, 7, 1, 4,
    4, 4, 8, 8, 3, 4,
    1, 5, 1, 2, 5, 2,
    6, 8, 3, 2, 7, 2};

// The tiles are:
// 2 2      9 7      1 4
// 4 4      8 8      3 4
//
// 1 5      1 2      5 2
// 6 8      3 2      7 2

// Averages.
int averagedata[] = {
    0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0,
};

array_view<int, 2> sample(4, 6, sampledata);
array_view<int, 2> average(4, 6, averagedata);

parallel_for_each(
    // Create threads for sample.grid and divide the grid into 2 x 2 tiles.
    sample.extent.tile<2,2>(),
    [=](tiled_index<2,2> idx) restrict(amp)
    {
        // Create a 2 x 2 array to hold the values in this tile.
        tile_static int nums[2][2];
        // Copy the values for the tile into the 2 x 2 array.
        nums[idx.local[1]][idx.local[0]] = sample[idx.global];
        // When all the threads have executed and the 2 x 2 array is complete, find the average.
        idx.barrier.wait();
        int sum = nums[0][0] + nums[0][1] + nums[1][0] + nums[1][1];
        // Copy the average into the array_view.
        average[idx.global] = sum / 4;
    }
);

for (int i = 0; i < 4; i++) {
    for (int j = 0; j < 6; j++) {
        std::cout << average(i,j) << " ";
    }
}

```

```
        }
        std::cout << "\n";
    }

// Output.
// 3 3 8 8 3 3
// 3 3 8 8 3 3
// 5 5 2 2 4 4
// 5 5 2 2 4 4
```

Confira também

[Modificadores específicos da Microsoft](#)

[Visão geral do C++ AMP](#)

[Função parallel_for_each \(C++ amp\)](#)

[Passo a passo: multiplicação de matrizes](#)

Específico da Microsoft

A sintaxe de atributo estendido para especificar informações de classe de armazenamento usa a `__declspec` palavra-chave, que especifica que uma instância de um determinado tipo deve ser armazenada com um atributo de classe de armazenamento específico da Microsoft listado abaixo. Exemplos de outros modificadores de classe de armazenamento incluem `static` e `extern` palavras-chave e, No entanto, essas palavras-chave fazem parte da especificação ANSI das linguagens C e C++ e, assim, não são abrangidas pela sintaxe de atributo estendido. A sintaxe de atributo estendido simplifica e padroniza extensões específicas da Microsoft para as linguagens C e C++.

Gramática

```
decl-specifier :  
  __declspec ( extended-decl-modifier-seq )  
  
extended-decl-modifier-seq :  
  extended-decl-modifier opt  
  extended-decl-modifier extended-decl-modifier-seq  
  
extended-decl-modifier :  
  align( ***número de** ) *  
  allocate(" segname ")  
  allocator  
  appdomain  
  code_seg(" segname ")  
  deprecated  
  dllimport  
  dllexport  
  jitintrinsic  
  naked  
  noalias  
  noinline  
  noreturn  
  nothrow  
  novtable  
  process  
  property( { get= Get-Func-Name| ,put= Put-Func-Name} )  
  restrict  
  safebuffers  
  selectany  
  spectre(nomitigation)  
  thread  
  uuid(" ComObjectGUID ")
```

O espaço em branco separa a sequência modificadora de declaração. Os exemplos aparecem nas seções posteriores.

A gramática de atributo estendido dá suporte a esses atributos de classe de armazenamento específicos da Microsoft: `align`, `allocate`, `allocator`, `appdomain`, `code_seg`, `deprecated`, `dllexport`, `dllimport`, `jitintrinsic`, `naked`, `noalias`, `noinline`, `noreturn`, `nothrow`, `novtable`, `process`, `restrict`, `safebuffers`, `selectany`, `spectre` e `thread`. Ele também dá suporte a esses atributos de objeto COM: `property` e `uuid`.

Os atributos `code_seg`, `dllexport` e de `dllimport`, `naked`, `noalias`, `nothrow`, `property`, `restrict`, `selectany`, `thread` e `uuid` classe de armazenamento são propriedades somente da declaração do objeto ou da função à qual eles são aplicados. O `thread` atributo afeta somente dados e objetos. Os `naked`, `spectre` atributos e afetam apenas funções. Os `dllimport`, `dllexport` atributos e afetam funções, dados e objetos. Os `property`, `selectany` atributos, e `uuid` afetam objetos com.

Para compatibilidade com versões anteriores, `_declspec` é um sinônimo para `_declspec`, a menos que a opção do compilador `/za` (desabilitar extensões de linguagem) seja especificada.

As `_declspec` palavras-chave devem ser colocadas no início de uma declaração simples. O compilador ignora, sem aviso, quaisquer `_declspec` palavras-chave colocadas após * ou & e na frente do identificador de variável em uma declaração.

Um `_declspec` atributo especificado no início de uma declaração de tipo definida pelo usuário se aplica à variável desse tipo. Por exemplo:

```
_declspec(dllexport) class X {} varX;
```

Nesse caso, o atributo se aplica a `varX`. Um `_declspec` atributo colocado depois de `class` ou `struct` palavra-chave ou se aplica ao tipo definido pelo usuário. Por exemplo:

```
class _declspec(dllexport) X {};
```

Nesse caso, o atributo se aplica a `X`.

A diretriz geral para usar o `_declspec` atributo para declarações simples é a seguinte:

decl-especificador-Seq init-declarator-List

O *decl-especificador-Seq* deve conter, entre outras coisas, um tipo base (por exemplo, `int`, `float` a `typedef`, ou um nome de classe), uma classe de armazenamento (por exemplo, `static`, `extern`) ou a `_declspec` extensão. A *lista init-declarator* deve conter, entre outras coisas, a parte do ponteiro das declarações. Por exemplo:

```
_declspec(selectany) int * pi1 = 0; //Recommended, selectany & int both part of decl-specifier
int _declspec(selectany) * pi2 = 0; //OK, selectany & int both part of decl-specifier
int * _declspec(selectany) pi3 = 0; //ERROR, selectany is not part of a declarator
```

O código a seguir declara uma variável local de thread inteiro e a inicializa com um valor:

```
// Example of the _declspec keyword
_declspec( thread ) int tls_i = 1;
```

FINAL específico da Microsoft

Consulte também

[Palavras-chave](#)

Atributos de classe de armazenamento estendido C

align (C++)

02/09/2020 • 15 minutes to read • [Edit Online](#)

No Visual Studio 2015 e posterior, use o especificador standard do C++ 11 `alignas` para controlar o alinhamento. Para obter mais informações, consulte [Alignment](#).

Específico da Microsoft

Use `__declspec(align(#))` para controlar precisamente o alinhamento dos dados definidos pelo usuário (por exemplo, alocações estáticas ou dados automáticos em uma função).

Sintaxe

Declarador de `__declspec (align (#)) declarator`

Comentários

Criar aplicativos que usam as instruções mais recentes do processador acarreta algumas restrições e problemas novos. Muitas novas instruções exigem dados alinhados a limites de 16 bytes. Além disso, ao alinhar os dados usados com frequência ao tamanho da linha de cache do processador, você melhora o desempenho do cache. Por exemplo, se você definir uma estrutura cujo tamanho é menor que 32 bytes, talvez queira o alinhamento de 32 bytes para garantir que os objetos desse tipo de estrutura sejam armazenados em cache de forma eficiente.

#é o valor de alinhamento. As entradas válidas são potências inteiras de dois de 1 a 8192 (bytes), como 2, 4, 8, 16, 32 ou 64. `declarator` são os dados que você está declarando como alinhados.

Para obter informações sobre como retornar um valor do tipo `size_t` que é o requisito de alinhamento do tipo, consulte [alignof](#). Para obter informações sobre como declarar ponteiros não alinhados ao direcionar processadores de 64 bits, consulte [_unaligned](#).

Você pode usar `__declspec(align(#))` quando define um `struct`, `union` ou `class`, ou quando você declara uma variável.

O compilador não garante ou tenta preservar o atributo de alinhamento dos dados durante uma operação de cópia ou de transformação de dados. Por exemplo, `memcpy` pode copiar uma `struct` declarada com `__declspec(align(#))` para qualquer local. Os alocadores comuns (por exemplo, `malloc` C++ `operator new` e os alocadores do Win32) normalmente retornam memória que não está suficientemente alinhada para `__declspec(align(#))` estruturas ou matrizes de estruturas. Para garantir que o destino de uma operação de cópia ou de transformação de dados esteja alinhado corretamente, use [_aligned_malloc](#). Ou escreva seu próprio alocador.

Você não pode especificar o alinhamento para parâmetros de função. Quando você passa dados que têm um atributo de alinhamento por valor na pilha, seu alinhamento é controlado pela Convenção de chamada. Se o alinhamento de dados for importante na função chamada, copie o parâmetro na memória alinhada corretamente antes de usar.

Sem `__declspec(align(#))`, o compilador geralmente alinha os dados em limites naturais com base no processador de destino e no tamanho dos dados, limites de até 4 bytes em processadores de 32 bits e limites de 8 bytes em processadores de 64 bits. Os dados em classes ou estruturas são alinhados na classe ou estrutura no mínimo de seu alinhamento natural e na configuração de empacotamento atual (de `#pragma pack` ou a `/zp` opção do compilador).

Este exemplo mostra o uso de `__declspec(align(#))`:

```
__declspec(align(32)) struct Str1{
    int a, b, c, d, e;
};
```

Esse tipo agora tem um atributo de alinhamento de 32 bytes. Isso significa que todas as instâncias estática e automática iniciam em um limite de 32 bytes. Tipos de estrutura adicionais declarados com esse tipo como um membro preservam o atributo de alinhamento deste tipo, ou seja, qualquer estrutura com `Str1` como um elemento tem um atributo de alinhamento de pelo menos 32.

Aqui, `sizeof(struct Str1)` é igual a 32. Isso significa que, se uma matriz de `Str1` objetos for criada e a base da matriz estiver alinhada em 32 bytes, cada membro da matriz também será de 32 bytes alinhado. Para criar uma matriz cuja base está alinhada corretamente na memória dinâmica, use `_aligned_malloc`. Ou escreva seu próprio alocador.

O `sizeof` valor de qualquer estrutura é o deslocamento do membro final, além do tamanho do membro, arredondado para o múltiplo mais próximo do maior valor de alinhamento de membro ou o valor de alinhamento de estrutura inteiro, o que for maior.

O compilador usa essas regras para alinhamento de estrutura:

- A menos que seja substituído por `__declspec(align(#))`, o alinhamento de um membro de estrutura escalar é o mínimo do seu tamanho e da compactação atual.
- A menos que seja substituído por `__declspec(align(#))`, o alinhamento de uma estrutura é o máximo de alinhamentos individuais dos seus membros.
- Um membro de estrutura é colocado em um deslocamento do início de sua estrutura pai que é o menor múltiplo de seu alinhamento maior ou igual ao deslocamento do final do membro anterior.
- O tamanho de uma estrutura é o menor múltiplo de seu maior alinhamento maior que ou igual ao deslocamento do final de seu último membro.

`__declspec(align(#))` só pode aumentar as restrições de alinhamento.

Para obter mais informações, consulte:

- [align Disso](#)
- [Definindo novos tipos com `__declspec\(align\(#\)\)`](#)
- [Alinhando dados no armazenamento local de threads](#)
- [Como o `align` funciona com a compactação de dados](#)
- [Exemplos de alinhamento de estrutura](#) (específico para x64)

alinear exemplos

Os exemplos a seguir mostram como `__declspec(align(#))` afeta o tamanho e o alinhamento de estruturas de dados. Os exemplos assumem as seguintes definições:

```
#define CACHE_LINE 32
#define CACHE_ALIGN __declspec(align(CACHE_LINE))
```

Neste exemplo, a estrutura `S1` é definida usando `__declspec(align(32))`. Todos os usos de `S1` para uma definição de variável ou em outras declarações de tipo são alinhados como 32 bytes. `sizeof(struct S1)` retorna

32 e `s1` tem 16 bytes de preenchimento depois dos 16 bytes necessários para armazenar os quatro inteiros. Cada `int` membro requer alinhamento de 4 bytes, mas o alinhamento da estrutura em si é declarado como 32. Em seguida, o alinhamento geral é 32.

```
struct CACHE_ALIGN S1 { // cache align all instances of S1
    int a, b, c, d;
};

struct S1 s1; // s1 is 32-byte cache aligned
```

Neste exemplo, `sizeof(struct S2)` retorna 16, que é exatamente a soma dos tamanhos do membro, pois esse é um múltiplo do maior requisito de alinhamento (um múltiplo de 8).

```
__declspec(align(8)) struct S2 {
    int a, b, c, d;
};
```

No exemplo a seguir, `sizeof(struct S3)` retorna 64.

```
struct S3 {
    struct S1 s1; // S3 inherits cache alignment requirement
                   // from S1 declaration
    int a;         // a is now cache aligned because of s1
                   // 28 bytes of trailing padding
};
```

Neste exemplo, observe que `a` tem o alinhamento de seu natural tipo, nesse caso, 4 bytes. No entanto, `s1` deve ser alinhado para 32 bytes. A seguir, 28 bytes de preenchimento `a`, para que `s1` comece no deslocamento 32. `s4` em seguida, herda o requisito de alinhamento do `s1`, porque ele é o maior requisito de alinhamento na estrutura. `sizeof(struct S4)` retorna 64.

```
struct S4 {
    int a;
    // 28 bytes padding
    struct S1 s1; // S4 inherits cache alignment requirement of S1
};
```

As três declarações de variável a seguir também usam `__declspec(align(#))`. Em cada caso, a variável deve ser alinhada para 32 bytes. Na matriz, o endereço base da matriz, e não cada membro da matriz, é alinhado em 32 bytes. O `sizeof` valor de cada membro da matriz não é afetado quando você usa o `__declspec(align(#))`.

```
CACHE_ALIGN int i;
CACHE_ALIGN int array[128];
CACHE_ALIGN struct s2 s;
```

Para alinhar cada membro de uma matriz, códigos como estes deverão ser usados:

```
typedef CACHE_ALIGN struct { int a; } S5;
S5 array[10];
```

Neste exemplo, observe que alinhar a estrutura em si e alinhar o primeiro elemento tem o mesmo efeito:

```
CACHE_ALIGN struct S6 {
    int a;
    int b;
};

struct S7 {
    CACHE_ALIGN int a;
    int b;
};
```

`S6` e `S7` têm alinhamento, alocação e características de tamanho idênticos.

Neste exemplo, o alinhamento dos endereços iniciais de `a`, `b`, `c` e `d` são 4, 1, 4 e 1, respectivamente.

```
void fn() {
    int a;
    char b;
    long c;
    char d[10]
}
```

O alinhamento, se a memória tiver sido alocada no heap, depende de qual função de alocação é chamada. Por exemplo, se você usar `malloc`, o resultado dependerá do tamanho do operando. Se $arg \geq 8$, a memória retornada será alinhada em 8 bytes. Se $arg < 8$, o alinhamento da memória retornado será a primeira potência de 2 menos de ARG . Por exemplo, se você usar `malloc(7)`, o alinhamento será de 4 bytes.

Definindo novos tipos com `__declspec(align(#))`

Você pode definir um tipo com uma característica de alinhamento.

Por exemplo, você pode definir um `struct` com um valor de alinhamento desta forma:

```
struct aType {int a; int b;};
typedef __declspec(align(32)) struct aType bType;
```

Agora, `aType` e `bType` têm o mesmo tamanho (8 bytes), mas as variáveis do tipo `bType` são alinhadas em 32 bytes.

Alinhando dados no armazenamento local de threads

O armazenamento local de thread estático (TLS) criado com o atributo `__declspec(thread)` e colocado na seção TLS na imagem funciona para o alinhamento exatamente como dados estáticos normais. Para criar dados TLS, o sistema operacional aloca memória do tamanho da seção do TLS e respeita o atributo de alinhamento da seção do TLS.

Este exemplo mostra várias maneiras de colocar dados alinhados no armazenamento local de threads.

```

// put an aligned integer in TLS
__declspec(thread) __declspec(align(32)) int a;

// define an aligned structure and put a variable of the struct type
// into TLS
__declspec(thread) __declspec(align(32)) struct F1 { int a; int b; } a;

// create an aligned structure
struct CACHE_ALIGN S9 {
    int a;
    int b;
};

// put a variable of the structure type into TLS
__declspec(thread) struct S9 a;

```

Como o `align` funciona com a compactação de dados

A `/Zp` opção do compilador e o `pack` pragma têm o efeito de empacotar dados para membros de estrutura e União. Este exemplo mostra como `/Zp` e `__declspec(align(#))` trabalhar juntos:

```

struct S {
    char a;
    short b;
    double c;
    CACHE_ALIGN double d;
    char e;
    double f;
};

```

A tabela a seguir lista o deslocamento de cada membro em `/Zp` valores diferentes (ou `#pragma pack`), mostrando como os dois interagem.

VARIÁVEL	<code>/ZP1</code>	<code>/ZP2</code>	<code>/ZP4</code>	<code>/ZP8</code>
um	0	0	0	0
b	1	2	2	2
c	3	4	4	8
d	32	32	32	32
e	40	40	40	40
f	41	42	44	48
<code>sizeof(S)</code>	64	64	64	64

Para obter mais informações, consulte [/Zp \(alinhamento de membro de struct\)](#).

O deslocamento de um objeto baseia-se no deslocamento do objeto anterior e na configuração atual de compactação, a menos que o objeto tenha um atributo `__declspec(align(#))`. Nesse caso, o alinhamento será baseado no deslocamento do objeto anterior e no valor de `__declspec(align(#))` para o objeto.

FINAL específico da Microsoft

Confira também

[_declspec](#)

[Visão geral das convenções da ABI do ARM](#)

[Convenções de software x64](#)

allocate

02/09/2020 • 2 minutes to read • [Edit Online](#)

Específico da Microsoft

O `allocate` especificador de declaração nomeia um segmento de dados no qual o item de dados será alocado.

Sintaxe

```
* __declspec(allocate("***segname")) Declarador
```

Comentários

O nome *segname* deve ser declarado usando um dos seguintes pragmas:

- [code_seg](#)
- [const_seg](#)
- [data_seg](#)
- [init_seg](#)
- [Section](#)

Exemplo

```
// allocate.cpp
#pragma section("mycode", read)
__declspec(allocate("mycode")) int i = 0;

int main() {
```

FINAL específico da Microsoft

Confira também

[__declspec](#)
Palavras-chave

allocator

02/09/2020 • 2 minutes to read • [Edit Online](#)

Específico da Microsoft

O `allocator` especificador de declaração pode ser aplicado a funções de alocação de memória personalizadas para tornar as alocações visíveis por meio do ETW (rastreamento de eventos para Windows).

Sintaxe

```
__declspec(allocator)
```

Comentários

O criador de perfil de memória nativa no Visual Studio funciona coletando dados de evento ETW de alocação emitidos pelo durante o tempo de execução. Os alocadores no CRT e no SDK do Windows foram anotados no nível de origem para que seus dados de alocação possam ser capturados. Se você estiver escrevendo seus próprios alocadores, todas as funções que retornam um ponteiro para a memória de heap alocada recentemente podem ser decoradas com `__declspec(allocator)`, como visto neste exemplo para `mymalloc`:

```
__declspec(allocator) void* myMalloc(size_t size)
```

Para obter mais informações, consulte [medir o uso de memória no Visual Studio](#) e eventos de [heap de ETW nativo personalizado](#).

FINAL específico da Microsoft

appdomain

25/03/2020 • 4 minutes to read • [Edit Online](#)

Especifica que cada domínio de seu aplicativo gerenciado deve ter sua própria cópia de uma variável global específica ou variável de membro estático. Consulte [domínios de aplicativo e C++ Visual](#) para obter mais informações.

Cada domínio de aplicativo tem sua própria cópia de uma variável per-appdomain. Um construtor de uma variável appdomain é executado quando um assembly é carregado em um domínio de aplicativo, e o destruidor é executado quando o domínio de aplicativo é descarregado.

Se você quiser que todos os domínios de aplicativo em um processo no Common Language Runtime compartilhem uma variável global, use o modificador `__declspec(process)`. `__declspec(process)` está em vigor por padrão em [/CLR](#). As opções de compilador [/CLR: Pure](#) e [/CLR: safe](#) são preteridas no Visual Studio 2015 e sem suporte no Visual Studio 2017.

`__declspec(appdomain)` só é válida quando uma das opções do compilador [/CLR](#) é usada. Somente uma variável global, uma variável de membro estática ou uma variável local estática pode ser marcada com `__declspec(appdomain)`. É um erro aplicar `__declspec(appdomain)` a membros estáticos de tipos gerenciados porque eles sempre têm esse comportamento.

O uso de `__declspec(appdomain)` é semelhante ao uso [do armazenamento local de threads \(TLS\)](#). Os threads têm seu próprio armazenamento, assim como os domínios de aplicativo. Usar `__declspec(appdomain)` garante que a variável global tenha seu próprio armazenamento em cada domínio de aplicativo criado para esse aplicativo.

Há limitações para misturar o uso de por processo e por variáveis de AppDomain; consulte [processo](#) para obter mais informações.

Por exemplo, na inicialização do programa, todas as variáveis per-process são inicializadas, depois todas as variáveis per-appdomain são inicializadas. Portanto, quando uma variável per-process está sendo inicializada, ela não pode depender do valor de qualquer variável de domínio per-appdomain. Não é uma boa prática combinar o uso (a atribuição) de variáveis per-appdomain e per-process.

Para obter informações sobre como chamar uma função em um domínio de aplicativo específico, consulte [Call_in_appdomain função](#).

Exemplo

```
// declspec_appdomain.cpp
// compile with: /clr
#include <stdio.h>
using namespace System;

class CGlobal {
public:
    CGlobal(bool bProcess) {
        Counter = 10;
        m_bProcess = bProcess;
        Console::WriteLine("__declspec({0}) CGlobal::CGlobal constructor", m_bProcess ? (String^)"process" :
(String^)"appdomain");
    }

    ~CGlobal() {
        Console::WriteLine("__declspec({0}) CGlobal::~CGlobal destructor", m_bProcess ? (String^)"process" :
(String^)"appdomain");
    }
}
```

```

}

int Counter;

private:
    bool m_bProcess;
};

__declspec(process) CGlobal process_global = CGlobal(true);
__declspec(appdomain) CGlobal appdomain_global = CGlobal(false);

value class Functions {
public:
    static void change() {
        ++appdomain_global.Counter;
    }

    static void display() {
        Console::WriteLine("process_global value in appdomain '{0}': {1}",
                           AppDomain::CurrentDomain->FriendlyName,
                           process_global.Counter);

        Console::WriteLine("appdomain_global value in appdomain '{0}': {1}",
                           AppDomain::CurrentDomain->FriendlyName,
                           appdomain_global.Counter);
    }
};

int main() {
    AppDomain^ defaultDomain = AppDomain::CurrentDomain;
    AppDomain^ domain = AppDomain::CreateDomain("Domain 1");
    AppDomain^ domain2 = AppDomain::CreateDomain("Domain 2");
    CrossAppDomainDelegate^ changeDelegate = gcnew CrossAppDomainDelegate(&Functions::change);
    CrossAppDomainDelegate^ displayDelegate = gcnew CrossAppDomainDelegate(&Functions::display);

    // Print the initial values of appdomain_global in all appdomains.
    Console::WriteLine("Initial value");
    defaultDomain->DoCallBack(displayDelegate);
    domain->DoCallBack(displayDelegate);
    domain2->DoCallBack(displayDelegate);

    // Changing the value of appdomain_global in the domain and domain2
    // appdomain_global value in "default" appdomain remain unchanged
    process_global.Counter = 20;
    domain->DoCallBack(changeDelegate);
    domain2->DoCallBack(changeDelegate);
    domain2->DoCallBack(changeDelegate);

    // Print values again
    Console::WriteLine("Changed value");
    defaultDomain->DoCallBack(displayDelegate);
    domain->DoCallBack(displayDelegate);
    domain2->DoCallBack(displayDelegate);

    AppDomain::Unload(domain);
    AppDomain::Unload(domain2);
}

```

```
__declspec(process) CGlobal::CGlobal constructor
__declspec(appdomain) CGlobal::CGlobal constructor
Initial value
process_global value in appdomain 'declspec_appdomain.exe': 10
appdomain_global value in appdomain 'declspec_appdomain.exe': 10
__declspec(appdomain) CGlobal::CGlobal constructor
process_global value in appdomain 'Domain 1': 10
appdomain_global value in appdomain 'Domain 1': 10
__declspec(appdomain) CGlobal::CGlobal constructor
process_global value in appdomain 'Domain 2': 10
appdomain_global value in appdomain 'Domain 2': 10
Changed value
process_global value in appdomain 'declspec_appdomain.exe': 20
appdomain_global value in appdomain 'declspec_appdomain.exe': 10
process_global value in appdomain 'Domain 1': 20
appdomain_global value in appdomain 'Domain 1': 11
process_global value in appdomain 'Domain 2': 20
appdomain_global value in appdomain 'Domain 2': 12
__declspec(appdomain) CGlobal::~CGlobal destructor
__declspec(appdomain) CGlobal::~CGlobal destructor
__declspec(appdomain) CGlobal::~CGlobal destructor
__declspec(process) CGlobal::~CGlobal destructor
```

Confira também

[__declspec](#)

[Palavras-chave](#)

code_seg (_declspec)

25/03/2020 • 6 minutes to read • [Edit Online](#)

Seção específica da Microsoft

O atributo **code_seg** declaração nomeia um segmento de texto executável no arquivo. obj no qual o código do objeto para as funções de membro de classe ou função será armazenado.

Sintaxe

```
_declspec(code_seg("segname")) declarator
```

Comentários

O atributo `_declspec(code_seg(...))` permite o posicionamento do código em segmentos nomeados distintos que podem ser paginados ou bloqueados na memória individualmente. Você pode usar esse atributo para controlar o posicionamento de modelos instanciados e do código gerado pelo compilador.

Um *segmento* é um bloco nomeado de dados em um arquivo. obj que é carregado na memória como uma unidade. Um *segmento de texto* é um segmento que contém código executável. A *seção* de termo geralmente é usada de forma intercambiável com o segmento.

O código de objeto gerado quando `declarator` é definido é colocado no segmento de texto especificado por `segname`, que é um literal de cadeia de caracteres estreito. O nome `segname` não precisa ser especificado em um pragma de `seção` antes que possa ser usado em uma declaração. Por padrão, quando nenhum `code_seg` é especificado, o código do objeto é colocado em um segmento chamado `.text`. Um atributo `code_seg` substitui qualquer diretiva `#pragma code_seg` existente. Um atributo `code_seg` aplicado a uma função de membro substitui qualquer atributo `code_seg` aplicado à classe delimitadora.

Se uma entidade tiver um atributo `code_seg`, todas as declarações e definições da mesma entidade deverão ter atributos de `code_seg` idênticos. Se uma classe base tiver um atributo `code_seg`, as classes derivadas deverão ter o mesmo atributo.

Quando um atributo `code_seg` é aplicado a uma função de escopo de namespace ou a uma função de membro, o código do objeto para essa função é colocado no segmento de texto especificado. Quando esse atributo é aplicado a uma classe, todas as funções membro da classe e classes aninhadas (isso inclui funções membro especiais geradas por compilador) são colocadas no segmento especificado. Classes definidas localmente — por exemplo, classes definidas em um corpo de função membro — não herdam o atributo `code_seg` do escopo delimitador.

Quando um atributo `code_seg` é aplicado a uma classe de modelo ou função de modelo, todas as especializações implícitas do modelo são colocadas no segmento especificado. Especializações explícitas ou parciais não herdam o atributo `code_seg` do modelo primário. Você pode especificar o mesmo atributo de `code_seg` ou diferente na especialização. Um atributo `code_seg` não pode ser aplicado a uma instânciação de modelo explícita.

Por padrão, o código gerado por compilador, como uma função membro especial, é colocado no segmento `.text`. A diretiva `#pragma code_seg` não substitui esse padrão. Use o atributo `code_seg` na classe, modelo de classe ou modelo de função para controlar onde o código gerado pelo compilador é colocado.

As lambdas herdam atributos de `code_seg` de seu escopo delimitador. Para especificar um segmento para um lambda, aplique um atributo `code_seg` após a cláusula de declaração de parâmetro e antes de qualquer especificação mutável ou de exceção, qualquer especificação de tipo de retorno à direita e o corpo lambda. Para

obter mais informações, consulte [sintaxe de expressão lambda](#). Este exemplo define uma lambda em um segmento denominado PagedMem:

```
auto Sqr = [](int t) __declspec(code_seg("PagedMem")) -> int { return t*t; };
```

Tenha cuidado ao colocar funções membro específicas, especialmente funções membro virtuais, em diferentes segmentos. Se você definir uma função virtual em uma classe derivada que resida em um segmento paginado quando o método da classe base residir em um segmento não paginado, outros métodos de classe base ou o código de usuário poderão supor que invocando o método virtual não acionarão uma falha de página.

Exemplo

Este exemplo mostra como um atributo **code_seg** controla o posicionamento de segmento quando a especialização de modelo implícita e explícita é usada:

```
// code_seg.cpp
// Compile: cl /EHsc /W4 code_seg.cpp

// Base template places object code in Segment_1 segment
template<class T>
class __declspec(code_seg("Segment_1")) Example
{
public:
    virtual void VirtualMemberFunction(T /*arg*/) {}

// bool specialization places code in default .text segment
template<>
class Example<bool>
{
public:
    virtual void VirtualMemberFunction(bool /*arg*/) {}

// int specialization places code in Segment_2 segment
template<>
class __declspec(code_seg("Segment_2")) Example<int>
{
public:
    virtual void VirtualMemberFunction(int /*arg*/) {}

// Compiler warns and ignores __declspec(code_seg("Segment_3"))
// in this explicit specialization
__declspec(code_seg("Segment_3")) Example<short>; // C4071

int main()
{
    // implicit double specialization uses base template's
    // __declspec(code_seg("Segment_1")) to place object code
    Example<double> doubleExample{};
    doubleExample.VirtualMemberFunction(3.14L);

    // bool specialization places object code in default .text segment
    Example<bool> boolExample{};
    boolExample.VirtualMemberFunction(true);

    // int specialization uses __declspec(code_seg("Segment_2"))
    // to place object code
    Example<int> intExample{};
    intExample.VirtualMemberFunction(42);
}
```

Fim da seção específica da Microsoft

Confira também

[_declspec](#)

[Palavras-chave](#)

deprecated (C++)

02/09/2020 • 2 minutes to read • [Edit Online](#)

Este tópico é sobre a declaração `declspec` preterida específica da Microsoft. Para obter informações sobre o atributo C++ 14 `[[deprecated]]` e orientação sobre quando usar esse atributo versus o `declspec` ou pragma específico da Microsoft, consulte [atributos padrão do C++](#).

Com as exceções indicadas abaixo, a `deprecated` declaração oferece a mesma funcionalidade que o pragma [preterido](#) :

- A `deprecated` declaração permite que você especifique formas específicas de sobrecargas de função como preteridas, enquanto o formulário pragma se aplica a todas as formas sobrecarregadas de um nome de função.
- A `deprecated` declaração permite que você especifique uma mensagem que será exibida no momento da compilação. O texto da mensagem pode ser de uma macro.
- As macros só podem ser marcadas como preteridas com o `deprecated` pragma.

Se o compilador encontrar o uso de um identificador substituído ou o atributo padrão `[[deprecated]]`, um aviso de [C4996](#) será gerado.

Exemplo

O exemplo a seguir mostra como marcar funções como preteridas e como especificar uma mensagem que será exibida no tempo de compilação, quando a função preterida for usada.

```
// deprecated.cpp
// compile with: /W3
#define MY_TEXT "function is deprecated"
void func1(void) {}
__declspec(deprecated) void func1(int) {}
__declspec(deprecated("** this is a deprecated function **")) void func2(int) {}
__declspec(deprecated(MY_TEXT)) void func3(int) {}

int main() {
    func1();
    func1(1);    // C4996
    func2(1);    // C4996
    func3(1);    // C4996
}
```

Exemplo

O exemplo a seguir mostra como marcar classes como preteridas e como especificar uma mensagem que será exibida no tempo de compilação, quando a classe preterida for usada.

```
// deprecate_class.cpp
// compile with: /W3
struct __declspec(deprecated) X {
    void f(){}
};

struct __declspec(deprecated("** X2 is deprecated **")) X2 {
    void f(){}
};

int main() {
    X x;      // C4996
    X2 x2;    // C4996
}
```

Confira também

[__declspec](#)

[Palavras-chave](#)

dllexport, dllimport

02/09/2020 • 3 minutes to read • [Edit Online](#)

Específico da Microsoft

O `__declspec(dllimport)` e os `__declspec(dllexport)` atributos de classe de armazenamento são extensões específicas da Microsoft para as linguagens C e C++. É possível usá-las para exportar e importar funções, dados e objetos para ou a partir de um DLL.

Sintaxe

```
__declspec( dllimport ) declarator
__declspec( dllexport ) declarator
```

Comentários

Esses atributos definem explicitamente a interface da DLL para seu cliente, que pode ser o arquivo executável ou outro DLL. A declaração de funções como `__declspec(dllexport)` elimina a necessidade de um arquivo de definição de módulo (. def), pelo menos em relação à especificação de funções exportadas. O `__declspec(dllexport)` atributo substitui a palavra-chave `__declspec(_export)`.

Se uma classe está marcada como `__declspec(dllexport)`, todas as especializações de modelos de classe na hierarquia de classe são marcadas implicitamente como `__declspec(dllexport)`. Isso significa que os modelos de classe são instanciados explicitamente e que os membros da classe devem ser definidos.

`__declspec(dllexport)` de uma função expõe a função com seu nome decorado. Para funções em C++, isso inclui a desconfiguração do nome. Para funções ou funções C declaradas como `extern "C"`, isso inclui a decoração específica da plataforma com base na Convenção de chamada. Para obter informações sobre a decoração de nome no código C/C++, consulte [nomes decorados](#). Nenhuma decoração de nome é aplicada às funções C exportadas ou às `extern "C"` funções C++ usando a `__cdecl` Convenção de chamada.

Para exportar um nome não decorado, você pode vincular usando um arquivo de definição de módulo (. def) que define o nome não decorado em uma seção exportações. Para obter mais informações, consulte [exportações](#).

Outra maneira de exportar um nome não decorado é usar uma

```
#pragma comment(linker, "/export:alias=decorated_name")
```

Ao declarar `__declspec(dllexport)` ou `__declspec(dllimport)`, você deve usar a [sintaxe de atributo estendido](#) e a `__declspec` palavra-chave.

Exemplo

```
// Example of the __declspec( dllimport ) and __declspec( dllexport ) class attributes
__declspec( dllimport ) int i;
__declspec( dllexport ) void func();
```

Como alternativa, para tornar seu código mais legível, você pode usar definições de macro:

```
#define DllImport  __declspec( dllexport )
#define DllExport   __declspec( dllimport )

DllExport void func();
DllExport int i = 10;
DllImport int j;
DllExport int n;
```

Para obter mais informações, consulte:

- [Definições e declarações](#)
- [Definindo funções C++ embutidas com dllexport e DllImport](#)
- [Regras e limitações gerais](#)
- [Usando DllImport e dllexport em classes C++](#)

FINAL específico da Microsoft

Confira também

[__declspec](#)

[Palavras-chave](#)

Definições e declarações (C++)

02/09/2020 • 2 minutes to read • [Edit Online](#)

Específico da Microsoft

A interface DLL refere-se a todos os itens (funções e dados) que são conhecidos por serem exportados por algum programa no sistema; ou seja, todos os itens que são declarados como `__declspec(dllimport)` ou `__declspec(dllexport)`. Todas as declarações incluídas na interface DLL devem especificar o `__declspec(dllimport)` atributo ou `__declspec(dllexport)`. No entanto, a definição deve especificar apenas o `__declspec(dllexport)` atributo. Por exemplo, a definição de função a seguir gera um erro de compilador:

```
__declspec(dllimport) int func() {    // Error; dllimport
                                    // prohibited on definition.
    return 1;
}
```

Este código também gera um erro:

```
__declspec(dllimport) int i = 10; // Error; this is a definition.
```

No entanto, esta é uma sintaxe correta:

```
__declspec(dllexport) int i = 10; // Okay--export definition
```

O uso de `__declspec(dllexport)` implica uma definição, enquanto `__declspec(dllimport)` implica uma declaração. Você deve usar a `extern` palavra-chave WITH `__declspec(dllexport)` para forçar uma declaração; caso contrário, uma definição é implícita. Assim, os seguintes exemplos estão corretos:

```
#define DllImport __declspec(dllimport)
#define DllExport __declspec(dllexport)

extern DllExport int k; // These are both correct and imply a
DllImport int j;      // declaration.
```

Os exemplos a seguir esclarecem o que foi dito acima:

```
static __declspec( dllexport ) int l; // Error; not declared extern.

void func() {
    static __declspec( dllimport ) int s; // Error; not declared
                                         // extern.
    __declspec( dllimport ) int m;      // Okay; this is a
                                         // declaration.
    __declspec( dllexport ) int n;     // Error; implies external
                                         // definition in local scope.
    extern __declspec( dllimport ) int i; // Okay; this is a
                                         // declaration.
    extern __declspec( dllexport ) int k; // Okay; extern implies
                                         // declaration.
    __declspec( dllexport ) int x = 5;  // Error; implies external
                                         // definition in local scope.
}
```

FINAL específico da Microsoft

Confira também

[dllexport, dllimport](#)

Definindo funções embutidas do C++ com `dllexport` e `dllimport`

02/09/2020 • 2 minutes to read • [Edit Online](#)

Específico da Microsoft

Você pode definir como embutido uma função com o `dllexport` atributo. Nesse caso, a função sempre é instanciada e exportada, mesmo se qualquer módulo no programa fizer referência à função. Presume-se que a função seja importada por outro programa.

Você também pode definir como embutida uma função declarada com o `dllimport` atributo. Nesse caso, a função pode ser expandida (sujeito às especificações de `/Ob`), mas nunca instanciada. Em particular, se o endereço de uma função importada embutida for usado, o endereço da função que reside na DLL será retornado. Esse comportamento é o mesmo que usar o endereço de uma função importada não embutida.

Essas regras se aplicam às funções embutidas cujas definições aparecem dentro de uma definição de classe. Além disso, os dados locais estáticos e as cadeias de caracteres em funções embutidas mantêm as mesmas identidades entre a DLL e o cliente como em um único programa (isto é, um arquivo executável sem uma interface de DLL).

Tenha cuidado ao fornecer funções embutidas importadas. Por exemplo, se você atualizar a DLL, não suponha que o cliente não usará a versão modificada da DLL. Para garantir que você esteja carregando a versão apropriada da DLL, recompile o cliente da DLL também.

FINAL específico da Microsoft

Confira também

[dllexport, dllimport](#)

Regras e limitações gerais

02/09/2020 • 5 minutes to read • [Edit Online](#)

Específico da Microsoft

- Se você declarar uma função ou um objeto sem `dllimport` ou `dllexport` atributo ou, a função ou o objeto não será considerado parte da interface DLL. Consequentemente, a definição da função ou do objeto deverá estar presente nesse módulo ou em outro módulo do mesmo programa. Para tornar a função ou objeto parte da interface DLL, você deve declarar a definição da função ou objeto no outro módulo como `dllexport`. Caso contrário, será gerado um erro de vinculador.

Se você declarar uma função ou um objeto com o `dllexport` atributo, sua definição deverá aparecer em algum módulo do mesmo programa. Caso contrário, será gerado um erro de vinculador.

- Se um único módulo em seu programa contiver `dllimport` `dllexport` declarações e para a mesma função ou objeto, o `dllexport` atributo terá precedência sobre o `dllimport` atributo. No entanto, um aviso do compilador será gerado. Por exemplo:

```
__declspec( dllimport ) int i;
__declspec( dllexport ) int i;  // Warning; inconsistent;
                             // dllexport takes precedence.
```

- Em C++, você pode inicializar um ponteiro de dados local declarado globalmente ou estático ou com o endereço de um objeto de dados declarado com o `dllimport` atributo, que gera um erro em C. Além disso, você pode inicializar um ponteiro de função local estático com o endereço de uma função declarada com o `dllimport` atributo. Em C, essa atribuição define o ponteiro como o endereço da conversão de importação da DLL (um stub de código que transfere o controle para a função) em vez do endereço da função. Em C++, ela define o ponteiro como o endereço da função. Por exemplo:

```
__declspec( dllimport ) void func1( void );
__declspec( dllimport ) int i;

int *pi = &i;                                // Error in C
static void ( *pf )( void ) = &func1;          // Address of thunk in C,
                                             // function in C++

void func2()
{
    static int *pi = &i;                      // Error in C
    static void ( *pf )( void ) = &func1; // Address of thunk in C,
                                         // function in C++
}
```

No entanto, como um programa que inclui o `dllexport` atributo na declaração de um objeto deve fornecer a definição para esse objeto em algum lugar no programa, você pode inicializar um ponteiro de função estática global ou local com o endereço de uma `dllexport` função. Da mesma forma, você pode inicializar um ponteiro de dados estático global ou local com o endereço de um `dllexport` objeto de dados. Por exemplo, o seguinte código não gera erros em C ou C++:

```

__declspec( dllexport ) void func1( void );
__declspec( dllexport ) int i;

int *pi = &i;                                // Okay
static void ( *pf )( void ) = &func1;          // Okay

void func2()
{
    static int *pi = &i;                      // Okay
    static void ( *pf )( void ) = &func1; // Okay
}

```

- Se você aplicar `dllexport` a uma classe regular que tenha uma classe base que não esteja marcada como `dllexport`, o compilador irá gerar C4275.

O compilador irá gerar o mesmo aviso se a classe base for uma especialização de um modelo de classe. Para contornar isso, marque a classe base com `dllexport`. O problema com uma especialização de um modelo de classe é onde colocá-lo `__declspec(dllexport)`; você não tem permissão para marcar o modelo de classe. Em vez disso, instancie explicitamente o modelo de classe e marque essa instanciação explícita com `dllexport`. Por exemplo:

```

template class __declspec(dllexport) B<int>;
class __declspec(dllexport) D : public B<int> {
// ...

```

Essa solução alternativa falhará se o argumento do modelo for a classe derivada. Por exemplo:

```

class __declspec(dllexport) D : public B<D> {
// ...

```

Como esse é um padrão comum com modelos, o compilador alterou a semântica de `dllexport` quando ele é aplicado a uma classe que tem uma ou mais classes base e quando uma ou mais das classes base é uma especialização de um modelo de classe. Nesse caso, o compilador se aplica implicitamente `dllexport` às especializações de modelos de classe. Você pode fazer o seguinte e não receber um aviso:

```

class __declspec(dllexport) D : public B<D> {
// ...

```

FINAL específico da Microsoft

Confira também

[dllexport, dllimport](#)

Usando o `dllimport` e o `dllexport` nas classes do C++

02/09/2020 • 7 minutes to read • [Edit Online](#)

Específico da Microsoft

Você pode declarar classes C++ com o `dllimport` `dllexport` atributo ou. Esses formulários implicam que a classe inteira será importada ou exportada. As classes exportadas dessa maneira são chamadas classes exportáveis.

O exemplo a seguir define uma classe exportável. Todas as suas funções de membro e dados estáticos são exportados:

```
#define DllExport __declspec( dllexport )

class DllExport C {
    int i;
    virtual int func( void ) { return 1; }
};
```

Observe que o uso explícito dos `dllimport` `dllexport` atributos e nos membros de uma classe exportável é proibido.

Classes `dllexport`

Quando você declara uma classe `dllexport`, todas as suas funções de membro e membros de dados estáticos são exportados. Você deve fornecer as definições de todos esses membros no mesmo programa. Caso contrário, será gerado um erro de vinculador. A única exceção a essa regra se aplica às funções virtuais puras, para as quais você não precisa fornecer definições explícitas. No entanto, como um destruidor de uma classe abstrata é sempre chamado pelo destruidor da classe base, os destruidores virtuais puros devem fornecer sempre uma definição. Essas regras são as mesmas para classes não exportáveis.

Se você exportar dados do tipo de classe ou de funções que retornarem classes, exporte a classe.

Classes `DllImport`

Quando você declara uma classe `dllimport`, todas as suas funções de membro e membros de dados estáticos são importados. Ao contrário do comportamento de `dllimport` e `dllexport` em tipos não de classe, os membros de dados estáticos não podem especificar uma definição no mesmo programa no qual uma `dllimport` classe é definida.

Classes de herança e exportáveis

Todas as classes base de uma classe exportável devem ser exportáveis. Caso contrário, um aviso do compilador será gerado. Além disso, todos os membros acessíveis que também são classes devem ser exportáveis. Essa regra permite `dllexport` que uma classe herde de uma `dllimport` classe e uma `dllimport` classe para herdar de uma `dllexport` classe (embora o último não seja recomendado). Como regra, tudo que pode ser acessado pelo cliente da DLL (de acordo com as regras de acesso do C++) deve fazer parte da interface exportável. Isso inclui os membros de dados confidenciais referenciados em funções integradas.

Importação/exportação de membro seletivo

Como as funções de membro e os dados estáticos em uma classe implicitamente têm ligação externa, você pode

declará-las com o `dllimport` `dllexport` atributo ou, a menos que toda a classe seja exportada. Se a classe inteira for importada ou exportada, a declaração explícita de funções e dados de membro como `dllimport` ou `dllexport` será proibida. Se você declarar um membro de dados estáticos dentro de uma definição de classe como `dllexport`, uma definição deverá ocorrer em algum lugar dentro do mesmo programa (como com vinculação externa sem classe).

Da mesma forma, você pode declarar funções de membro com os `dllimport` `dllexport` atributos ou. Nesse caso, você deve fornecer uma `dllexport` definição em algum lugar dentro do mesmo programa.

Vale a pena observar vários pontos importantes sobre a importação e a exportação seletiva de membros:

- A importação/exportação seletiva de membros é melhor usada para o fornecimento de uma versão da interface de classe exportada mais restritiva, isto é, uma para a qual você pode criar uma DLL que exponha menos recursos públicos e particulares do que a linguagem normalmente permitiria. Também é útil para ajustar a interface exportável: quando você souber que o cliente, por definição, não pode acessar alguns dados particulares, não precisará exportar a classe inteira.
- Se você exportar uma função virtual em uma classe, deverá exportar todas elas ou pelo menos fornecer versões que poderão ser diretamente usadas pelo cliente.
- Se você tiver uma classe em que usa a importação/exportação seletiva de membros com funções virtuais, as funções deverão estar na interface exportável ou ser definidas como integradas (podem ser vistas pelo cliente).
- Se você definir um membro como `dllexport`, mas não incluí-lo na definição de classe, um erro do compilador será gerado. Você deve definir o membro no cabeçalho da classe.
- Embora a definição de membros de classe como `dllimport` ou `dllexport` seja permitida, você não pode substituir a interface especificada na definição de classe.
- Se você definir uma função de membro em um local diferente do corpo da definição de classe na qual você o declarou, um aviso será gerado se a função for definida como `dllexport` ou `dllimport` (se essa definição for diferente da especificada na declaração de classe).

FINAL específico da Microsoft

Confira também

[dllexport](#), [dllimport](#)

jitintrinsic

02/09/2020 • 2 minutes to read • [Edit Online](#)

Marca a função como significante para o Common Language Runtime de 64 bits. Isso é usado em determinadas funções em bibliotecas fornecidas pela Microsoft.

Sintaxe

```
__declspec(jitintrinsic)
```

Comentários

`jitintrinsic` Adiciona um MODOPT ([IsJitIntrinsic](#)) a uma assinatura de função.

Os usuários são desencorajados de usar esse `__declspec` modificador, pois podem ocorrer resultados inesperados.

Confira também

[__declspec](#)

[Palavras-chave](#)

naked (C++)

02/09/2020 • 2 minutes to read • [Edit Online](#)

Específico da Microsoft

Para funções declaradas com o `naked` atributo, o compilador gera código sem prólogo e código epílogo. Você pode usar esse recurso para escrever suas próprias sequências de código de prólogo/epílogo usando o código de assembler embutido. As funções naked são particularmente úteis para escrever drivers para dispositivo virtuais. Observe que o `naked` atributo só é válido no x86 e no ARM e não está disponível em x64.

Sintaxe

```
__declspec(naked) declarator
```

Comentários

Como o `naked` atributo só é relevante para a definição de uma função e não é um modificador de tipo, as funções Naked devem usar a sintaxe de atributo estendido e a palavra-chave `__declspec`.

O compilador não pode gerar uma função embutida para uma função marcada com o atributo Naked, mesmo que a função também seja marcada com a palavra-chave `__forceinline`.

O compilador emitirá um erro se o `naked` atributo for aplicado a qualquer coisa que não seja a definição de um método que não seja membro.

Exemplos

Esse código define uma função com o `naked` atributo:

```
__declspec( naked ) int func( formal_parameters ) {}
```

Ou, alternativamente:

```
#define Naked __declspec( naked )
Naked int func( formal_parameters ) {}
```

O `naked` atributo afeta apenas a natureza da geração de código do compilador para as sequências de prólogo e epílogo da função. Não afeta o código que é gerado pela chamada dessas funções. Portanto, o `naked` atributo não é considerado parte do tipo da função e os ponteiros de função não podem ter o `naked` atributo. Além disso, o `naked` atributo não pode ser aplicado a uma definição de dados. Por exemplo, esta amostra de código gera um erro:

```
__declspec( naked ) int i;
// Error--naked attribute not permitted on data declarations.
```

O `naked` atributo é relevante apenas para a definição da função e não pode ser especificado no protótipo da função. Por exemplo, esta declaração gera um erro de compilador:

```
__declspec( naked ) int func(); // Error--naked attribute not permitted on function declarations
```

FINAL específico da Microsoft

Confira também

[__declspec](#)

[Palavras-chave](#)

[Chamadas de função Naked](#)

Específico da Microsoft

`noalias` significa que uma chamada de função não modifica ou referencia o estado global visível e modifica apenas a memória apontada *diretamente* por parâmetros de ponteiro (indireções de primeiro nível).

Se uma função for anotada como `noalias`, o otimizador poderá pressupor que apenas os próprios parâmetros e apenas as indireções de primeiro nível dos parâmetros de ponteiro serão referenciados ou modificados dentro da função.

A `noalias` anotação só se aplica dentro do corpo da função anotada. Marcar uma função como `__declspec(noalias)` não afeta o alias de ponteiros retornados pela função.

Para outra anotação que pode afetar o alias, consulte [__declspec\(restrict\)](#).

Exemplo

O exemplo a seguir demonstra o uso de `__declspec(noalias)`.

Quando a função `multiply` que acessa a memória é anotada `__declspec(noalias)`, ela informa ao compilador que essa função não modifica o estado global, exceto pelos ponteiros em sua lista de parâmetros.

```

// declspec_noalias.c
#include <stdio.h>
#include <stdlib.h>

#define M 800
#define N 600
#define P 700

float * mempool, * memptr;

float * ma(int size)
{
    float * retval;
    retval = memptr;
    memptr += size;
    return retval;
}

float * init(int m, int n)
{
    float * a;
    int i, j;
    int k=1;

    a = ma(m * n);
    if (!a) exit(1);
    for (i=0; i<m; i++)
        for (j=0; j<n; j++)
            a[i*n+j] = 0.1/k++;
    return a;
}

__declspec(noalias) void multiply(float * a, float * b, float * c)
{
    int i, j, k;

    for (j=0; j<P; j++)
        for (i=0; i<M; i++)
            for (k=0; k<N; k++)
                c[i * P + j] =
                    a[i * N + k] *
                    b[k * P + j];
}

int main()
{
    float * a, * b, * c;

    mempool = (float *) malloc(sizeof(float) * (M*N + N*P + M*P));

    if (!mempool)
    {
        puts("ERROR: Malloc returned null");
        exit(1);
    }

    memptr = mempool;
    a = init(M, N);
    b = init(N, P);
    c = init(M, P);

    multiply(a, b, c);
}

```

Confira também

[__declspec](#)

Palavras-chave

[__declspec\(restrict\)](#)

noinline

02/09/2020 • 2 minutes to read • [Edit Online](#)

Específico da Microsoft

`__declspec(noinline)` informa ao compilador para nunca embutir uma função de membro específica (função em uma classe).

Pode ser válido não embutir uma função quando ela é pequena e não é crítica para o desempenho do seu código. Ou seja, se a função for pequena e se for improvável que ela seja chamada com frequência, como uma função que trata de uma condição de erro.

Tenha em mente que, se uma função estiver marcada `noinline`, a função de chamada será menor e, portanto, uma candidata para a entrada de compilador.

```
class X {
    __declspec(noinline) int mbrfunc() {
        return 0;
    }    // will not inline
};
```

FINAL específico da Microsoft

Confira também

[__declspec](#)

[Palavras-chave](#)

[inline, __inline, __forceinline](#)

noreturn

02/09/2020 • 2 minutes to read • [Edit Online](#)

Específico da Microsoft

Esse `__declspec` atributo informa ao compilador que uma função não retorna. Como consequência, o compilador sabe que o código após uma chamada para uma `__declspec(noreturn)` função está inacessível.

Se o compilador encontra uma função com um caminho de controle que não retorna um valor, ele gera um aviso (C4715) ou uma mensagem de erro (C2202). Se o caminho de controle não puder ser alcançado devido a uma função que nunca retorna, você poderá usar `__declspec(noreturn)` para evitar esse aviso ou erro.

NOTE

Adicionar `__declspec(noreturn)` a uma função que deve retornar pode resultar em um comportamento indefinido.

Exemplo

No exemplo a seguir, a `else` cláusula não contém uma instrução `return`. Declarar `fatal` como `__declspec(noreturn)` evita uma mensagem de erro ou de aviso.

```
// noreturn2.cpp
__declspec(noreturn) extern void fatal () {}

int main() {
    if(1)
        return 1;
    else if(0)
        return 0;
    else
        fatal();
}
```

FINAL específico da Microsoft

Confira também

[__declspec](#)

[Palavras-chave](#)

nothrow (C++)

02/09/2020 • 2 minutes to read • [Edit Online](#)

Específico da Microsoft

Um `__declspec` atributo estendido que pode ser usado na declaração de funções.

Sintaxe

tipo de retorno __declspec (nothrow) [chamar-Convenção] nome-da-função ([argumento-lista])

Comentários

É recomendável que todo o novo código use o operador `noexcept` em vez de `__declspec(nothrow)`.

Esse atributo diz ao compilador que a função declarada, e as funções que ela chama nunca lançam uma exceção. No entanto, ele não impõe a diretiva. Em outras palavras, ele nunca faz com que `std::Terminate` seja invocado, diferentemente `noexcept` ou no modo `std: C++ 17` (Visual Studio 2017 versão 15,5 e posterior), `throw()`.

Com o modelo de tratamento de exceções síncronas, agora padrão, o compilador pode eliminar a mecânica de acompanhar o tempo de vida útil de determinados objetos desenroláveis nessa função e reduzir significativamente o tamanho do código. Dada a seguinte diretiva de pré-processador, as três declarações de função abaixo são equivalentes no modo `/std: c++ 14`:

```
#define WINAPI __declspec(nothrow) __stdcall

void WINAPI f1();
void __declspec(nothrow) __stdcall f2();
void __stdcall f3() throw();
```

Em `/std: modo c++ 17`, `throw()` não é equivalente aos outros que usam `__declspec(nothrow)` porque ele faz com que ele seja `std::terminate` invocado se uma exceção for lançada da função.

A `void __stdcall f3() throw();` declaração usa a sintaxe definida pelo padrão C++. Em C++ 17, a `throw()` palavra-chave foi preferida.

FINAL específico da Microsoft

Confira também

[__declspec](#)

[noexcept](#)

[Palavras-chave](#)

novtable

02/09/2020 • 2 minutes to read • [Edit Online](#)

Específico da Microsoft

Este é um `__declspec` atributo estendido.

Essa forma de `__declspec` pode ser aplicada a qualquer declaração de classe, mas só deve ser aplicada a classes de interface pura, ou seja, classes que nunca serão instanciadas por conta própria. O `__declspec` interrompe o compilador de gerar código para inicializar o vfptr no (s) Construtor (es) e no destruidor da classe. Em diversos casos, isso remove as únicas referências para vtable que estão associadas à classe. Portanto, o vinculador as removerá. O uso dessa forma de `__declspec` pode resultar em uma redução significativa no tamanho do código.

Se você tentar criar uma instância de uma classe marcada com `novtable` e, em seguida, acessar um membro de classe, receberá uma violação de acesso (AV).

Exemplo

```
// novtable.cpp
#include <stdio.h>

struct __declspec(novtable) X {
    virtual void mf();
};

struct Y : public X {
    void mf() {
        printf_s("In Y\n");
    }
};

int main() {
    // X *pX = new X();
    // pX->mf();    // Causes a runtime access violation.

    Y *pY = new Y();
    pY->mf();
}
```

In Y

FINAL específico da Microsoft

Confira também

[__declspec](#)

[Palavras-chave](#)

Especifica que o processo de aplicativo gerenciado deve ter uma única cópia de uma determinada variável global, variável de membro estática ou variável local estática compartilhada por todos os domínios de aplicativo no processo. Isso foi projetado principalmente para ser usado durante a compilação com `/clr:pure`, que foi preferido no visual studio 2015 e sem suporte no visual studio 2017. Ao compilar com `/clr`, as variáveis global e estática são por processo por padrão e não precisam usar `__declspec(process)`.

Somente uma variável global, uma variável de membro estático ou uma variável local estática do tipo nativo podem ser marcadas com `__declspec(process)`.

`process` Só é válida durante a compilação com `/clr`.

Se você quiser que cada domínio de aplicativo tenha sua própria cópia de uma variável global, use [AppDomain](#).

Consulte [domínios e Visual C++ do aplicativo](#) para obter mais informações.

Confira também

[__declspec](#)

[Palavras-chave](#)

property (C++)

25/03/2020 • 2 minutes to read • [Edit Online](#)

Seção específica da Microsoft

Esse atributo pode ser aplicado a “membros de dados virtuais” não estáticos em uma definição de classe ou estrutura. O compilador trata esses “membros de dados virtuais” como membros de dados alterando suas referências em chamadas de função.

Sintaxe

```
__declspec( property( get=get_func_name ) ) declarator
__declspec( property( put=put_func_name ) ) declarator
__declspec( property( get=get_func_name, put=put_func_name ) ) declarator
```

Comentários

Quando o compilador vê um membro de dados declarado com esse atributo à direita de um operador de seleção de membro (". " ou "-> "), ele converte a operação em uma função `get` ou `put`, dependendo se tal expressão é um l-Value ou um valor r. Em contextos mais complicados, como "`+=`", uma regravação é executada fazendo `get` e `put`.

Esse atributo também pode ser usado na declaração de uma matriz vazia em uma definição de classe ou estrutura. Por exemplo:

```
__declspec(property(get=GetX, put=PutX)) int x[];
```

A instrução anterior indica que `x[]` pode ser usado com um ou mais índices da matriz. Nesse caso, `i=p->x[a][b]` será transformado em `i=p->GetX(a, b)`, e `p->x[a][b] = i` será transformado em `p->PutX(a, b, i);`

Fim da seção específica da Microsoft

Exemplo

```
// declspec_property.cpp
struct S {
    int i;
    void putprop(int j) {
        i = j;
    }

    int getprop() {
        return i;
    }

    __declspec(property(get = getprop, put = putprop)) int the_prop;
};

int main() {
    S s;
    s.the_prop = 5;
    return s.the_prop;
}
```

Confira também

[__declspec](#)

[Palavras-chave](#)

restrict

02/09/2020 • 3 minutes to read • [Edit Online](#)

Específico da Microsoft

Quando aplicado a uma declaração ou definição de função que retorna um tipo de ponteiro, `restrict` diz ao compilador que a função retorna um objeto que não tem um *alias*, ou seja, referenciado por quaisquer outros ponteiros. Isso permite que o compilador execute otimizações adicionais.

Sintaxe

```
* __declspec(restrict) ***pointer_return_type funçãopointer_return_type();
```

Comentários

O compilador é propagado `__declspec(restrict)`. Por exemplo, a `malloc` função CRT tem uma `__declspec(restrict)` decoração e, portanto, o compilador supõe que os ponteiros inicializados para locais de memória `malloc` também não têm o alias de ponteiros existentes anteriormente.

O compilador não verifica se o ponteiro retornado não é, na verdade, alias. É responsabilidade do desenvolvedor garantir que o programa não faça o alias de um ponteiro marcado com o modificador `restrict __declspec`.

Para obter semântica semelhante em variáveis, consulte [__restrict](#).

Para outra anotação que se aplica a alias em uma função, consulte [__declspec \(noalias\)](#).

Para obter informações sobre a `restrict` palavra-chave que faz parte de C++ &, consulte [restringir \(C++ &\)](#).

Exemplo

O exemplo a seguir demonstra o uso de `__declspec(restrict)`.

Quando `__declspec(restrict)` é aplicado a uma função que retorna um ponteiro, isso informa ao compilador que a memória apontada pelo valor de retorno não tem um alias. Neste exemplo, os ponteiros `mempool` e `memptr` são globais, portanto, o compilador não pode ter certeza de que a memória à qual se refere não tem um alias. No entanto, eles são usados em `ma` e `init` em seu chamador de uma forma que retorna a memória que não é referenciada pelo programa, de modo que `__declspec (restrito)` é usado para ajudar o otimizador. Isso é semelhante a como os cabeçalhos CRT decoram funções de alocação, como `malloc` usando o `__declspec(restrict)` para indicar que sempre retornam memória que não pode ser alias por ponteiros existentes.

```

// declspec_restrict.c
// Compile with: cl /W4 declspec_restrict.c
#include <stdio.h>
#include <stdlib.h>

#define M 800
#define N 600
#define P 700

float * mempool, * memptr;

__declspec(restrict) float * ma(int size)
{
    float * retval;
    retval = memptr;
    memptr += size;
    return retval;
}

__declspec(restrict) float * init(int m, int n)
{
    float * a;
    int i, j;
    int k=1;

    a = ma(m * n);
    if (!a) exit(1);
    for (i=0; i<m; i++)
        for (j=0; j<n; j++)
            a[i*n+j] = 0.1f/k++;
    return a;
}

void multiply(float * a, float * b, float * c)
{
    int i, j, k;

    for (j=0; j<P; j++)
        for (i=0; i<M; i++)
            for (k=0; k<N; k++)
                c[i * P + j] =
                    a[i * N + k] *
                    b[k * P + j];
}

int main()
{
    float * a, * b, * c;

    mempool = (float *) malloc(sizeof(float) * (M*N + N*P + M*P));

    if (!mempool)
    {
        puts("ERROR: Malloc returned null");
        exit(1);
    }

    memptr = mempool;
    a = init(M, N);
    b = init(N, P);
    c = init(M, P);

    multiply(a, b, c);
}

```

Confira também

[Palavras-chave](#)

[__declspec](#)

[__declspec \(noalias\)](#)

safebuffers

02/09/2020 • 3 minutes to read • [Edit Online](#)

Específico da Microsoft

Diz para o compilador não inserir verificações de segurança de excesso de buffer para uma função.

Sintaxe

```
__declspec( safebuffers )
```

Comentários

A opção de compilador [/GS](#) faz com que o compilador teste as saturações de buffer inserindo verificações de segurança na pilha. Os tipos de estruturas de dados elegíveis para verificações de segurança são descritos em [/GS \(verificação de segurança do buffer\)](#). Para obter mais informações sobre a detecção de estouro de buffer, consulte [recursos de segurança no MSVC](#).

Uma análise de código manual por especialista ou uma análise externa pode determinar que a função está protegida contra o excesso de buffer. Nesse caso, você pode suprimir as verificações de segurança de uma função aplicando a `__declspec(safebuffers)` palavra-chave à declaração da função.

Caution

As verificações de segurança do buffer fornecem a proteção de segurança importante e têm uma influência insignificante no desempenho. Portanto, recomendamos que você não suprime, exceto em casos raros em que o desempenho de uma função for um problema crítico e a função é comprovadamente segura.

Funções embutidas

Uma *função primária* pode usar uma palavra-chave [inline](#) para inserir uma cópia de uma *função secundária*. Se a `__declspec(safebuffers)` palavra-chave for aplicada a uma função, a detecção de estouro de buffer será suprimida para essa função. No entanto, a inlinização afeta a `__declspec(safebuffers)` palavra-chave das seguintes maneiras.

Suponha que a opção de compilador [/GS](#) seja especificada para ambas as funções, mas a função Primary especifica a `__declspec(safebuffers)` palavra-chave. As estruturas de dados na função secundária a tornam elegível para verificações de segurança, e a função não suprime essas verificações. Nesse caso:

- Especifique a palavra-chave [_forceinline](#) na função secundária para forçar o compilador a embutir essa função, independentemente das otimizações do compilador.
- Como a função secundária está qualificada para verificações de segurança, as verificações de segurança também são aplicadas à função primária, embora ela especifique a `__declspec(safebuffers)` palavra-chave.

Exemplo

O código a seguir mostra como usar a `__declspec(safebuffers)` palavra-chave.

```
// compile with: /c /GS
typedef struct {
    int x[20];
} BUFFER;
static int checkBuffers() {
    BUFFER cb;
    // Use the buffer...
    return 0;
};
static __declspec(safebuffers)
int noCheckBuffers() {
    BUFFER ncb;
    // Use the buffer...
    return 0;
}
int wmain() {
    checkBuffers();
    noCheckBuffers();
    return 0;
}
```

FINAL específico da Microsoft

Confira também

[__declspec](#)

[Palavras-chave](#)

[inline, __inline, __forceinline](#)

[strict_gs_check](#)

Específico da Microsoft

Informa ao compilador que o item de dados global declarado (variável ou objeto) é uma COMDAT aleatória (uma função compactada).

Sintaxe

```
* __declspec( selectany ) ***Declarador
```

Comentários

No tempo de link, se várias definições de uma COMDAT forem consideradas, o vinculador escolherá uma e descartará o restante. Se a opção de vinculador [/OPT:REF](#) (otimizações) for selecionada, a eliminação COMDAT ocorrerá para remover todos os itens de dados não referenciados na saída do vinculador.

Os construtores e a atribuição pela função ou por métodos estáticos globais na declaração não criam uma referência e não impedem a eliminação de /OPT:REF. Os efeitos colaterais desse código não devem depender de quando não houver nenhuma outra referência aos dados.

Para objetos dinamicamente inicializados, o `selectany` também descartará um código de inicialização de objeto não referenciado.

Um item de dados global normalmente pode ser inicializado apenas uma vez em um projeto EXE ou DLL.

`selectany` pode ser usado na inicialização de dados globais definidos por cabeçalhos, quando o mesmo cabeçalho aparece em mais de um arquivo de origem. `selectany` está disponível nos compiladores C e C++.

NOTE

`selectany` Só pode ser aplicado à inicialização real de itens de dados globais que são visíveis externamente.

Exemplo

Este código mostra como usar o `selectany` atributo:

```

//Correct - x1 is initialized and externally visible
__declspec(selectany) int x1=1;

//Incorrect - const is by default static in C++, so
//x2 is not visible externally (This is OK in C, since
//const is not by default static in C)
const __declspec(selectany) int x2 =2;

//Correct - x3 is extern const, so externally visible
extern const __declspec(selectany) int x3=3;

//Correct - x4 is extern const, so it is externally visible
extern const int x4;
const __declspec(selectany) int x4=4;

//Incorrect - __declspec(selectany) is applied to the uninitialized
//declaration of x5
extern __declspec(selectany) int x5;

// OK: dynamic initialization of global object
class X {
public:
X(int i){i++;};
int i;
};

__declspec(selectany) X x(1);

```

Exemplo

Este código mostra como usar o `selectany` atributo para garantir que os dados sejam dobrados por COMDAT quando você também usa a `/OPT:ICF` opção de vinculador. Observe que os dados devem ser marcados com `selectany` e colocados em uma `const` seção (ReadOnly). Você deve especificar explicitamente a seção somente leitura.

```

// selectany2.cpp
// in the following lines, const marks the variables as read only
__declspec(selectany) extern const int ix = 5;
__declspec(selectany) extern const int jx = 5;
int main() {
    int ij;
    ij = ix + jx;
}

```

FINAL específico da Microsoft

Confira também

[__declspec](#)

[Palavras-chave](#)

spectre

02/12/2019 • 2 minutes to read • [Edit Online](#)

Seção específica da Microsoft

Instrui o compilador a não inserir instruções de barreira de execução especulativa de Spectre variante 1 para uma função.

Sintaxe

```
__declspec( spectre(nomitigation) )
```

Comentários

A opção de compilador [/Qspectre](#) faz com que o compilador insira instruções de barreira de execução especulativa. Eles são inseridos onde a análise indica que existe uma vulnerabilidade de segurança de Spectre variante 1. As instruções específicas emitidas dependem do processador. Embora essas instruções devam ter um impacto mínimo sobre o tamanho ou o desempenho do código, pode haver casos em que o código não é afetado pela vulnerabilidade e requer o desempenho máximo.

A análise de especialistas pode determinar que uma função é segura de um Spectre de limites de verificação de falha de desvio da variante 1. Nesse caso, você pode suprimir a geração de código de mitigação dentro de uma função `__declspec(spectre(nomitigation))` aplicando-se à declaração da função.

Caution

As instruções de barreira de execução especulativa de [/Qspectre](#) fornecem proteção de segurança importante e apresentam um impacto insignificante sobre o desempenho. Portanto, recomendamos que você não suprime, exceto em casos raros em que o desempenho de uma função for um problema crítico e a função é comprovadamente segura.

Exemplo

O código a seguir mostra como usar `__declspec(spectre(nomitigation))` o.

```
// compile with: /c /Qspectre
static __declspec(spectre(nomitigation))
int noSpectreIssues() {
    // No Spectre variant 1 vulnerability here
    // ...
    return 0;
}

int main() {
    noSpectreIssues();
    return 0;
}
```

Fim da seção específica da Microsoft

Consulte também

[__declspec](#)

Palavras-chave

/Qspectre

thread

02/09/2020 • 5 minutes to read • [Edit Online](#)

Específico da Microsoft

O `thread` modificador de classe de armazenamento estendido é usado para declarar uma variável local de thread. Para o equivalente portátil no C++ 11 e posterior, use o especificador de classe de armazenamento `thread_local` para código portátil. No Windows `thread_local` é implementado com o `__declspec(thread)`.

Sintaxe

* `__declspec(thread) ***Declarador`

Comentários

O TLS (armazenamento local de threads) é o mecanismo pelo qual cada thread em um processo multithread aloca armazenamento para dados específicos ao thread. Em programas multithread padrão, os dados são compartilhados entre todos os threads de um processo específico, enquanto o armazenamento local de threads é o mecanismo para alocar dados por thread. Para obter uma discussão completa sobre threads, consulte [multithreading](#).

As declarações de variáveis locais de thread devem usar a [sintaxe de atributo estendido](#) e a `__declspec` palavra-chave com a `thread` palavra-chave. Por exemplo, o código a seguir declara uma variável local de thread de inteiro e a inicializa com um valor:

```
__declspec( thread ) int tls_i = 1;
```

Ao usar variáveis de local de thread em bibliotecas carregadas dinamicamente, você precisa estar ciente dos fatores que podem fazer com que uma variável de local de thread não seja inicializada corretamente:

1. Se a variável for inicializada com uma chamada de função (incluindo construtores), essa função só será chamada para o thread que fez com que o binário/DLL fosse carregado no processo e para os threads iniciados depois que o binário/DLL foi carregado. As funções de inicialização não são chamadas para qualquer outro thread que já estava em execução quando a DLL foi carregada. A inicialização dinâmica ocorre na chamada DllMain para DLL_THREAD_ATTACH, mas a DLL nunca receberá essa mensagem se a DLL não estiver no processo quando o thread for iniciado.
2. Variáveis locais de thread que são inicializadas estaticamente com valores constantes geralmente são inicializadas corretamente em todos os threads. No entanto, a partir de dezembro de 2017, há um problema de conformidade conhecido no compilador do Microsoft C++, pelo qual as `constexpr` variáveis recebem inicialização dinâmica em vez de estática.

Observação: os dois problemas devem ser corrigidos em atualizações futuras do compilador.

Além disso, você deve observar essas diretrizes ao declarar variáveis e objetos locais de thread:

- Você pode aplicar o `thread` atributo somente às declarações e definições de classe e de dados; `thread` não pode ser usado em declarações ou definições de função.
- Você pode especificar o `thread` atributo somente em itens de dados com duração de armazenamento estático. Isso inclui objetos de dados globais (`static` e `extern`), objetos estáticos locais e membros de

dados estáticos de classes. Você não pode declarar objetos de dados automáticos com o `thread` atributo.

- Você deve usar o `thread` atributo para a declaração e a definição de um objeto local de thread, se a declaração e a definição ocorrem no mesmo arquivo ou em arquivos separados.
- Você não pode usar o `thread` atributo como um modificador de tipo.
- Como a declaração de objetos que usam o `thread` atributo é permitida, esses dois exemplos são semanticamente equivalentes:

```
// declspec_thread_2.cpp
// compile with: /LD
__declspec( thread ) class B {
public:
    int data;
} BObject;    // BObject declared thread local.

class B2 {
public:
    int data;
};
__declspec( thread ) B2 BObject2;    // BObject2 declared thread local.
```

- O C padrão permite a inicialização de um objeto ou variável com uma expressão que envolve uma referência a si só, mas somente para objetos não estáticos. Embora o C++ normalmente permita essa inicialização dinâmica de um objeto com uma expressão que envolva uma referência a si mesmo, esse tipo de inicialização não é permitido com objetos locais de thread. Por exemplo:

```
// declspec_thread_3.cpp
// compile with: /LD
#define Thread __declspec( thread )
int j = j;    // Okay in C++; C error
Thread int tls_i = sizeof( tls_i );    // Okay in C and C++
```

Uma `sizeof` expressão que inclui o objeto que está sendo inicializado não constitui uma referência a si mesma e é permitida em C e C++.

FINAL específico da Microsoft

Confira também

[__declspec](#)

[Palavras-chave](#)

[Armazenamento local de threads \(TLS\)](#)

uuid (C++)

02/09/2020 • 2 minutes to read • [Edit Online](#)

Específico da Microsoft

O compilador anexa um GUID a uma classe ou estrutura declarada ou definida (somente definições de objeto COM completas) com o `uuid` atributo.

Sintaxe

```
__declspec( uuid("ComObjectGUID") ) declarator
```

Comentários

O `uuid` atributo usa uma cadeia de caracteres como seu argumento. Essa cadeia de caracteres nomeia um GUID no formato normal do registro com ou sem os delimitadores {} . Por exemplo:

```
struct __declspec(uuid("00000000-0000-0000-c000-000000000046")) IUnknown;
struct __declspec(uuid("{00020400-0000-0000-c000-000000000046}) IDispatch;
```

Esse atributo poderá ser aplicado a uma redeclaração. Isso permite que os cabeçalhos do sistema forneçam as definições de interfaces, como `IUnknown` , e a redeclaração em algum outro cabeçalho (como <comdef.h>) para fornecer o GUID.

A palavra-chave `__uuidof` pode ser aplicada para recuperar o GUID de constante anexado a um tipo definido pelo usuário.

FINAL específico da Microsoft

Confira também

[__declspec](#)

[Palavras-chave](#)

__restrict

02/09/2020 • 2 minutes to read • [Edit Online](#)

Assim como o modificador `__declspec (restrict)`, a `__restrict` palavra-chave indica que um símbolo não tem um alias no escopo atual. A `__restrict` palavra-chave é diferente do `__declspec (restrict)` modificador das seguintes maneiras:

- A `__restrict` palavra-chave é válida somente em variáveis e `__declspec (restrict)` só é válida em declarações de função e definições.
- `__restrict` é semelhante à `restrict` da especificação C99, mas `__restrict` pode ser usado em programas C++ ou C.
- Quando `__restrict` for usado, o compilador não propagará a propriedade sem alias de uma variável. Ou seja, se você atribuir uma `__restrict` variável a uma não `__restrict` variável, o compilador ainda permitirá que a variável não `__restrict` seja alias. Isso é diferente do comportamento da `restrict` palavra-chave da especificação C99.

Em geral, se você afetar o comportamento de uma função inteira, será melhor usar `__declspec (restrict)` do que a palavra-chave.

Para compatibilidade com versões anteriores, `restrict` é um sinônimo para `__restrict` a menos que a opção do compilador `/za (desabilitar extensões de linguagem)` seja especificada.

No Visual Studio 2015 e posterior, o `__restrict` pode ser usado em referências do C++.

NOTE

Quando usado em uma variável que também tem a palavra-chave `volatile`, `volatile` terá precedência.

Exemplo

```
// __restrict_keyword.c
// compile with: /LD
// In the following function, declare a and b as disjoint arrays
// but do not have same assurance for c and d.
void sum2(int n, int * __restrict a, int * __restrict b,
          int * c, int * d) {
    int i;
    for (i = 0; i < n; i++) {
        a[i] = b[i] + c[i];
        c[i] = b[i] + d[i];
    }
}

// By marking union members as __restrict, tell compiler that
// only z.x or z.y will be accessed in any given scope.
union z {
    int * __restrict x;
    double * __restrict y;
};
```

Confira também

Palavras-chave

__sptr, __uptr

02/09/2020 • 4 minutes to read • [Edit Online](#)

Específico da Microsoft

Use o `__sptr` `__uptr` modificador ou em uma declaração de ponteiro de 32 bits para especificar como o compilador converte um ponteiro de 32 bits para um ponteiro de 64 bits. Um ponteiro de 32 bits é convertido, por exemplo, quando ele é atribuído a uma variável de ponteiro de 64 bits ou sua referência é cancelada em uma plataforma de 64 bits.

A documentação da Microsoft para o suporte de plataformas de 64 bits às vezes chama o bit mais significativo de um ponteiro de 32 bits de bit de sinal. Por padrão, o compilador usa a extensão de sinal para converter um ponteiro de 32 bits em um ponteiro de 64 bits. Ou seja, os 32 bits menos significativos do ponteiro de 64 bits são definidos como o valor do ponteiro de 32 bits e os 32 bits mais significativos são definidos como o valor do bit de sinal do ponteiro de 32 bits. Essa conversão gera resultados corretos se o bit de sinal for 0, mas não se o bit de sinal for 1. Por exemplo, o endereço de 32 bits 0x7FFFFFFF produz o endereço de 64 bits 0x000000007FFFFFFF equivalente, mas o endereço de 32 bits 0x80000000 é alterado incorretamente para 0xFFFFFFFF80000000.

O `__sptr` modificador, ou ponteiro assinado, especifica que uma conversão de ponteiro definiu os bits mais significativos de um ponteiro de 64 bits para o bit de sinal do ponteiro de 32 bits. O `__uptr` modificador, ou não assinado, especifica que uma conversão definiu os bits mais significativos como zero. As declarações a seguir mostram `__sptr` os `__uptr` modificadores e usados com dois ponteiros não qualificados, dois ponteiros qualificados com o tipo de `__ptr32` e um parâmetro de função.

```
int * __sptr psp;
int * __uptr pup;
int * __ptr32 __sptr psp32;
int * __ptr32 __uptr pup32;
void MyFunction(char * __uptr __ptr32 myValue);
```

Use os `__sptr` `__uptr` modificadores e com declarações de ponteiro. Use os modificadores na posição de um [qualificador de tipo de ponteiro](#), o que significa que o modificador deve seguir o asterisco. Você não pode usar os modificadores com [ponteiros para membros](#). Os modificadores não afetam as declarações que não são de ponteiro.

Para compatibilidade com versões anteriores, `sptr` e `uptr` são sinônimos de `__sptr` e `__uptr`, a menos que a opção do compilador [/za \(desabilitar extensões de idioma\)](#) seja especificada.

Exemplo

O exemplo a seguir declara os ponteiros de 32 bits que usam os `__sptr` `__uptr` modificadores e, atribui cada ponteiro de 32 bits a uma variável de ponteiro de 64 bits e, em seguida, exibe o valor hexadecimal de cada ponteiro de 64 bits. O exemplo é criado com o compilador nativo de 64 bits e executado em uma plataforma de 64 bits.

```

// sptr_uptr.cpp
// processor: x64
#include <stdio.h>

int main()
{
    void *      __ptr64 p64;
    void *      __ptr32 p32d; //default signed pointer
    void * __sptr __ptr32 p32s; //explicit signed pointer
    void * __uptr __ptr32 p32u; //explicit unsigned pointer

    // Set the 32-bit pointers to a value whose sign bit is 1.
    p32d = reinterpret_cast<void *>(0x87654321);
    p32s = p32d;
    p32u = p32d;

    // The printf() function automatically displays leading zeroes with each 32-bit pointer. These are unrelated
    // to the __sptr and __uptr modifiers.
    printf("Display each 32-bit pointer (as an unsigned 64-bit pointer):\n");
    printf("p32d:      %p\n", p32d);
    printf("p32s:      %p\n", p32s);
    printf("p32u:      %p\n", p32u);

    printf("\nDisplay the 64-bit pointer created from each 32-bit pointer:\n");
    p64 = p32d;
    printf("p32d: p64 = %p\n", p64);
    p64 = p32s;
    printf("p32s: p64 = %p\n", p64);
    p64 = p32u;
    printf("p32u: p64 = %p\n", p64);
    return 0;
}

```

```

Display each 32-bit pointer (as an unsigned 64-bit pointer):
p32d:      000000087654321
p32s:      000000087654321
p32u:      000000087654321

```

```

Display the 64-bit pointer created from each 32-bit pointer:
p32d: p64 = FFFFFFFF87654321
p32s: p64 = FFFFFFFF87654321
p32u: p64 = 000000087654321

```

FINAL específico da Microsoft

Confira também

[Modificadores específicos da Microsoft](#)

`_unaligned`

02/09/2020 • 2 minutes to read • [Edit Online](#)

Específico da Microsoft. Quando você declara um ponteiro com o `_unaligned` modificador, o compilador pressupõe que o ponteiro endereça os dados que não estão alinhados. Consequentemente, o código apropriado para a plataforma é gerado para lidar com leituras e gravações não alinhadas pelo ponteiro.

Comentários

Este modificador descreve o alinhamento dos dados endereçados pelo ponteiro; o ponteiro em si é considerado alinhado.

A necessidade da `_unaligned` palavra-chave varia de acordo com a plataforma e o ambiente. A falha na marcação dos dados de forma adequada pode resultar em problemas que variam de penalidades de desempenho a falhas de hardware. O `_unaligned` modificador não é válido para a plataforma x86.

Para compatibilidade com versões anteriores, `_unaligned` é um sinônimo para `_unaligned` a menos que a opção do compilador `/za` ([desabilite extensões de linguagem](#)) seja especificada.

Para obter mais informações sobre alinhamento, consulte:

- [align](#)
- [alignof Operador](#)
- [pack](#)
- [/zp \(Alinhamento de Membro struct\)](#)
- [Exemplos de alinhamento da estrutura](#)

Confira também

[Palavras-chave](#)

__w64

02/09/2020 • 3 minutes to read • [Edit Online](#)

Esta palavra-chave específica da Microsoft está obsoleta. Em versões do Visual Studio anteriores à Visual Studio 2013, isso permite que você marque as variáveis, de modo que, quando você compilar com [/Wp64](#), o compilador relatará quaisquer avisos que seriam informados se você estivesse compilando com um compilador de 64 bits.

Sintaxe

tipo `__w64` de *identificador* do

parâmetros

tipo

Um dos três tipos que poderia causar problemas no código que está sendo portado de um de 32 bits para um compilador de 64 bits: `int`, `long` ou um ponteiro.

ID

O identificador da variável que você está criando.

Comentários

IMPORTANT

A opção de compilador [/Wp64](#) e a `__w64` palavra-chave são preteridas no Visual Studio 2010 e Visual Studio 2013 e removidas a partir do Visual Studio 2013. Se você usar a `/Wp64` opção do compilador na linha de comando, o compilador emitirá o aviso de linha de comando D9002. A `__w64` palavra-chave é silenciosamente ignorada. Em vez de usar essa opção e palavra-chave para detectar problemas de portabilidade de 64 bits, use um compilador do Microsoft C++ que tenha como alvo uma plataforma de 64 bits. Para obter mais informações, consulte [configurar Visual C++ para destinos de 64 bits, x64](#).

Qualquer TypeDef que tenha `__w64` nele deve ser 32 bits em x86 e 64 bits em x64.

Para detectar problemas de portabilidade usando versões do compilador do Microsoft C++ anteriores ao Visual Studio 2010, a `__w64` palavra-chave deve ser especificada em quaisquer TYPEDEFs que alteram o tamanho entre plataformas de 32 bits e 64 bits. Para qualquer tipo como esse, `__w64` deve aparecer apenas na definição de 32 bits do typedef.

Para compatibilidade com versões anteriores, `__w64` é um sinônimo para `__w64` a menos que a opção do compilador [/za \(desabilitar extensões de linguagem\)](#) seja especificada.

A `__w64` palavra-chave será ignorada se a compilação não usar `/Wp64`.

Para obter mais informações sobre portabilidade para 64 bits, consulte os tópicos a seguir:

- [Opções do compilador MSVC](#)
- [Portabilidade de código de 32 bits para código de 64 bits](#)
- [Configurar o Visual C++ para destinos x64 de 64 bits](#)

Exemplo

```
// __w64.cpp
// compile with: /W3 /WP64
typedef int Int_32;
#ifndef _WIN64
typedef __int64 Int_Native;
#else
typedef int __w64 Int_Native;
#endif

int main() {
    Int_32 i0 = 5;
    Int_Native i1 = 10;
    i0 = i1;    // C4244 64-bit int assigned to 32-bit int

    // char __w64 c;  error, cannot use __w64 on char
}
```

Confira também

[Palavras-chave](#)

func

25/03/2020 • 2 minutes to read • [Edit Online](#)

(C++ 11) O identificador `__predefinido__ Func` é implicitamente definido como uma cadeia de caracteres que contém o nome não qualificado e não adornado da função de circunscrição. `__Func__` é obrigatório pelo C++ padrão e não é uma extensão da Microsoft.

Sintaxe

```
__func__
```

Valor retornado

Retorna uma matriz de caracteres `const` com final de nulo que contém o nome da função.

Exemplo

```
#include <string>
#include <iostream>

namespace Test
{
    struct Foo
    {
        static void DoSomething(int i, std::string s)
        {
            std::cout << __func__ << std::endl; // Output: DoSomething
        }
    };
}

int main()
{
    Test::Foo::DoSomething(42, "Hello");

    return 0;
}
```

Requisitos

C++11

Supporte para COM do compilador

02/09/2020 • 2 minutes to read • [Edit Online](#)

Específico da Microsoft

O compilador do Microsoft C++ pode ler diretamente bibliotecas de tipo COM (modelo de objeto de componente) e converter o conteúdo em código-fonte C++ que pode ser incluído na compilação. As extensões de linguagem estão disponíveis para facilitar a programação COM no lado do cliente para aplicativos da área de trabalho.

Usando a [diretiva de pré-processador #import](#), o compilador pode ler uma biblioteca de tipos e convertê-la em um arquivo de cabeçalho C++ que descreve as interfaces com como classes. Um conjunto de atributos de `#import` está disponível para o usuário controlar o conteúdo referente aos arquivos resultantes de cabeçalho de biblioteca de tipos.

Você pode usar o [UUID](#) de atributo estendido `_declspec` para atribuir um GUID (identificador global exclusivo) a um objeto com. A palavra-chave `_uuidof` pode ser usada para extrair o GUID associado a um objeto com. Outro `_declspec` atributo, [Propriedade](#), pode ser usado para especificar os `get` `set` métodos e para um membro de dados de um objeto com.

Um conjunto de funções e classes globais de suporte de COM é fornecido para dar suporte aos `VARIANT` `BSTR` tipos e, implementar ponteiros inteligentes e encapsular o objeto de erro gerado por `_com_raise_error` :

- [Funções globais COM do compilador](#)
- [_bstr_t](#)
- [_com_error](#)
- [_com_ptr_t](#)
- [_variant_t](#)

FINAL específico da Microsoft

Confira também

[Classes de suporte de COM do compilador](#)

[Funções globais COM do compilador](#)

Funções globais COM do compilador

25/03/2020 • 2 minutes to read • [Edit Online](#)

Seção específica da Microsoft

As seguintes rotinas estão disponíveis:

FUNÇÃO	Descrição
_com_raise_error	Gera um _com_error em resposta a uma falha.
_set_com_error_handler	Substitui a função padrão que é usada para o tratamento de erros COM.
ConvertBSTRToString	Converte um valor <code>BSTR</code> em uma <code>char *</code> .
ConvertStringToBSTR	Converte um valor <code>char *</code> em uma <code>BSTR</code> .

Fim da seção específica da Microsoft

Confira também

[Classes de suporte COM do compilador](#)

[Suporte para COM do compilador](#)

_com_raise_error

22/04/2020 • 2 minutes to read • [Edit Online](#)

Específico da Microsoft

Lança um [_com_error](#) em resposta a uma falha.

Sintaxe

```
void __stdcall _com_raise_error(
    HRESULT hr,
    IErrorInfo* perrinfo = 0
);
```

Parâmetros

hr

Informações do HRESULT.

perrinfo

Objeto [IErrorInfo](#).

Comentários

`_com_raise_error`, que <é definido em `comdef.h`>, pode ser substituído por uma versão escrita pelo usuário com o mesmo nome e protótipo. Isso poderia ser feito se você quisesse usar `#import`, mas não quisesse usar ao tratamento de exceções do C++. Nesse caso, uma versão `_com_raise_error` do usuário do `longjmp` `_com_raise_error` pode decidir fazer uma ou exibir uma caixa de mensagem e parar. No entanto, a versão do usuário não deve retornar, pois o código de suporte do compilador COM não espera que ela retorne.

Você também pode usar [_set_com_error_handler](#) para substituir a função padrão de manipulação de erros.

Por padrão, `_com_raise_error` é definido da seguinte forma:

```
void __stdcall _com_raise_error(HRESULT hr, IErrorInfo* perrinfo) {
    throw _com_error(hr, perrinfo);
}
```

Fim específico da Microsoft

Requisitos

Cabeçalho: <comdef.h>

Lib: Se a opção de compilador **de tipo nativo wchar_t** estiver ativada, use comsuppw.lib ou comsuppwd.lib. Se **wchar_t** é **Tipo Nativo** está desligado, use comsupp.lib. Para obter mais informações, consulte [/Zc:wchar_t](#) ([wchar_t](#) é o tipo nativo).

Confira também

[Funções globais COM do compilador](#)

[_set_com_error_handler](#)

ConvertStringToBSTR

25/03/2020 • 2 minutes to read • [Edit Online](#)

Seção específica da Microsoft

Converte um valor `char *` em uma `BSTR`.

Sintaxe

```
BSTR __stdcall ConvertStringToBSTR(const char* pSrc)
```

parâmetros

pSrc

Uma variável `char *`.

Exemplo

```
// ConvertStringToBSTR.cpp
#include <comutil.h>
#include <stdio.h>

#pragma comment(lib, "comsuppw.lib")
#pragma comment(lib, "kernel32.lib")

int main() {
    char* lpszText = "Test";
    printf_s("char * text: %s\n", lpszText);

    BSTR bstrText = _com_util::ConvertStringToBSTR(lpszText);
    wprintf_s(L"BSTR text: %s\n", bstrText);

    SysFreeString(bstrText);
}
```

```
char * text: Test
BSTR text: Test
```

Fim da seção específica da Microsoft

Requisitos

Cabeçalho: `<fileutil.h>`

Lib: `comsuppw.lib` ou `comsuppwd.lib` (consulte [/Zc: Wchar_t \(Wchar_t é o tipo nativo\)](#) para obter mais informações)

Confira também

[Funções globais COM do compilador](#)

ConvertBSTRToString

25/03/2020 • 2 minutes to read • [Edit Online](#)

Seção específica da Microsoft

Converte um valor `BSTR` em uma `char *`.

Sintaxe

```
char* __stdcall ConvertBSTRToString(BSTR pSrc);
```

parâmetros

pSrc

Uma variável BSTR.

Comentários

`ConvertBSTRToString` aloca uma cadeia de caracteres que você deve excluir.

Exemplo

```
// ConvertBSTRToString.cpp
#include <comutil.h>
#include <stdio.h>

#pragma comment(lib, "comsuppw.lib")

int main() {
    BSTR bstrText = ::SysAllocString(L"Test");
    wprintf_s(L"BSTR text: %s\n", bstrText);

    char* lpszText2 = _com_util::ConvertBSTRToString(bstrText);
    printf_s("char * text: %s\n", lpszText2);

    SysFreeString(bstrText);
    delete[] lpszText2;
}
```

```
BSTR text: Test
char * text: Test
```

Fim da seção específica da Microsoft

Requisitos

Cabeçalho: <fileutil.h >

Lib: comsuppw.lib ou comsuppwd.lib (consulte [/Zc: Wchar_t \(Wchar_t é o tipo nativo\)](#) para obter mais informações)

Confira também

_set_com_error_handler

22/04/2020 • 2 minutes to read • [Edit Online](#)

Substitui a função padrão que é usada para o tratamento de erros COM. `_set_com_error_handler` é específico da Microsoft.

Sintaxe

```
void __stdcall _set_com_error_handler(
    void (__stdcall *pHandler)(
        HRESULT hr,
        IErrorInfo* perrinfo
    )
);
```

Parâmetros

pHandler

Ponteiro para a função de substituição.

hr

Informações do HRESULT.

perrinfo

Objeto `IErrorInfo`.

Comentários

Por padrão, `_com_raise_error` lida com todos os erros de COM. Você pode alterar esse comportamento usando `_set_com_error_handler` para chamar sua própria função de manipulação de erros.

A função de substituição deve ter uma assinatura equivalente à de `_com_raise_error`.

Exemplo

```

// _set_com_error_handler.cpp
// compile with /EHsc
#include <stdio.h>
#include <comdef.h>
#include <comutil.h>

// Importing ado dll to attempt to establish an ado connection.
// Not related to _set_com_error_handler
#import "C:\Program Files\Common Files\System\ado\msado15.dll" no_namespace rename("EOF", "adoEOF")

void __stdcall _My_com_raise_error(HRESULT hr, IErrorInfo* perrinfo)
{
    throw "Unable to establish the connection!";
}

int main()
{
    _set_com_error_handler(_My_com_raise_error);
    _bstr_t bstrEmpty(L"");
    _ConnectionPtr Connection = NULL;
    try
    {
        Connection.CreateInstance(__uuidof(Connection));
        Connection->Open(bstrEmpty, bstrEmpty, bstrEmpty, 0);
    }
    catch(char* errorMessage)
    {
        printf("Exception raised: %s\n", errorMessage);
    }

    return 0;
}

```

Exception raised: Unable to establish the connection!

Requisitos

Cabeçalho: <comdef.h>

Lib: Se a opção de compilador **/Zc:wchar_t** for especificada (o padrão), use comsuppw.lib ou comsuppwd.lib. Se a opção **/Zc:wchar_t**- compilador for especificada, use comsupp.lib. Para obter mais informações, incluindo como definir essa opção no IDE, consulte [/Zc:wchar_t \(wchar_t É o Tipo Nativo\)](#).

Confira também

[Funções globais COM do compilador](#)

Classes de suporte COM do compilador

25/03/2020 • 2 minutes to read • [Edit Online](#)

Seção específica da Microsoft

As classes padrão são usadas para dar suporte a alguns dos tipos COM. As classes são definidas em `<comdef.h>` e os arquivos de cabeçalho gerados na biblioteca de tipos.

CLASSE	FINALIDADE
<code>_bstr_t</code>	Encapsula o tipo <code>BSTR</code> para fornecer operadores e métodos úteis.
<code>_com_error</code>	Define o objeto de erro gerado por <code>_com_raise_error</code> na maioria das falhas.
<code>_com_ptr_t</code>	Encapsula ponteiros de interface COM e automatiza as chamadas necessárias para <code>AddRef</code> , <code>Release</code> e <code>QueryInterface</code> .
<code>_variant_t</code>	Encapsula o tipo <code>VARIANT</code> para fornecer operadores e métodos úteis.

Fim da seção específica da Microsoft

Confira também

[Suporte para COM do compilador](#)

[Funções globais COM do compilador](#)

[Referência da linguagem C++](#)

Classe _bstr_t

02/09/2020 • 2 minutes to read • [Edit Online](#)

Específico da Microsoft

Um `_bstr_t` objeto encapsula o [tipo de dados BSTR](#). A classe gerencia a alocação de recursos e a desalocação por meio de chamadas de função para `SysAllocString` e `SysFreeString` e outras `BSTR` APIs quando apropriado. A classe `_bstr_t` usa a contagem de referência para evitar sobrecarga excessiva.

Construção

CONSTRUTOR	DESCRIÇÃO
<code>_bstr_t</code>	Constrói um objeto <code>_bstr_t</code> .

Operations

FUNÇÃO	DESCRIÇÃO
<code>Assign</code>	Copia um <code>BSTR</code> para o <code>BSTR</code> encapsulado por um <code>_bstr_t</code> .
<code>Attach</code>	Vincula um wrapper <code>_bstr_t</code> a um <code>BSTR</code> .
<code>copy</code>	Constrói uma cópia do <code>BSTR</code> encapsulado.
<code>Desanexar</code>	Retorna o <code>BSTR</code> encapsulado por um <code>_bstr_t</code> e desanexa o <code>BSTR</code> do <code>_bstr_t</code> .
<code>GetAddress</code>	Aponta para o <code>BSTR</code> encapsulado por um <code>_bstr_t</code> .
<code>GetBSTR</code>	Aponta para o início do <code>BSTR</code> encapsulado por <code>_bstr_t</code> .
<code>length</code>	Retorna o número de caracteres no <code>_bstr_t</code> .

Operadores

OPERADOR	DESCRIÇÃO
<code>operador =</code>	Atribui um novo valor a um objeto <code>_bstr_t</code> existente.
<code>operador +=</code>	Acrescenta caracteres ao final do objeto <code>_bstr_t</code> .
<code>operador +</code>	Concatena duas cadeias de caracteres.
<code>operador !=</code>	Verifica se o encapsulado <code>BSTR</code> é uma cadeia de caracteres nula.
<code>operador ==, !=, <, >, <=, >=</code>	Compara dois objetos <code>_bstr_t</code> .

OPERADOR	DESCRIÇÃO
wchar_t do operador * Char*	Extrai os ponteiros para o objeto BSTR Unicode ou multibyte encapsulado.

FINAL específico da Microsoft

Requisitos

Cabeçalho: <comutil.h>

Lib: comsuppw.lib ou comsuppwd.lib (consulte [/Zc: Wchar_t \(Wchar_t é o tipo nativo\)](#) para obter mais informações)

Confira também

[Classes de suporte de COM do compilador](#)

Funções de membro `_bstr_t`

25/03/2020 • 2 minutes to read • [Edit Online](#)

Para obter informações sobre `_bstr_t` funções membro, consulte [_Bstr_t classe](#).

Confira também

[Classe `_bstr_t`](#)

_bstr_t::Assign

02/09/2020 • 2 minutes to read • [Edit Online](#)

Específico da Microsoft

Copia um `BSTR` para o `BSTR` encapsulado por um `_bstr_t`.

Sintaxe

```
void Assign(  
    BSTR s  
>);
```

Parâmetros

`&`

Um `BSTR` a ser copiado para o `BSTR` encapsulado por um `_bstr_t`.

Comentários

Atribuir uma cópia binária, o que significa que todo o comprimento do `BSTR` é copiado, independentemente do conteúdo.

Exemplo

```

// _bstr_t_Assign.cpp

#include <comdef.h>
#include <stdio.h>

int main()
{
    // creates a _bstr_t wrapper
    _bstr_t bstrWrapper;

    // creates BSTR and attaches to it
    bstrWrapper = "some text";
    wprintf_s(L"bstrWrapper = %s\n",
             static_cast<wchar_t*>(bstrWrapper));

    // bstrWrapper releases its BSTR
    BSTR bstr = bstrWrapper.Detach();
    wprintf_s(L"bstrWrapper = %s\n",
             static_cast<wchar_t*>(bstrWrapper));
    // "some text"
    wprintf_s(L"bstr = %s\n", bstr);

    bstrWrapper.Attach(SysAllocString(OLESTR("SysAllocatedString")));
    wprintf_s(L"bstrWrapper = %s\n",
             static_cast<wchar_t*>(bstrWrapper));

    // assign a BSTR to our _bstr_t
    bstrWrapper.Assign(bstr);
    wprintf_s(L"bstrWrapper = %s\n",
             static_cast<wchar_t*>(bstrWrapper));

    // done with BSTR, do manual cleanup
    SysFreeString(bstr);

    // reuse bstr
    bstr= SysAllocString(OLESTR("Yet another string"));
    // two wrappers, one BSTR
    _bstr_t bstrWrapper2 = bstrWrapper;

    *bstrWrapper.GetAddress() = bstr;

    // bstrWrapper and bstrWrapper2 do still point to BSTR
    bstr = 0;
    wprintf_s(L"bstrWrapper = %s\n",
             static_cast<wchar_t*>(bstrWrapper));
    wprintf_s(L"bstrWrapper2 = %s\n",
             static_cast<wchar_t*>(bstrWrapper2));

    // new value into BSTR
    _snwprintf_s(bstrWrapper.GetBSTR(), 100, bstrWrapper.length(),
                 L"changing BSTR");
    wprintf_s(L"bstrWrapper = %s\n",
             static_cast<wchar_t*>(bstrWrapper));
    wprintf_s(L"bstrWrapper2 = %s\n",
             static_cast<wchar_t*>(bstrWrapper2));
}

```

```
bstrWrapper = some text
bstrWrapper = (null)
bstr = some text
bstrWrapper = SysAllocedString
bstrWrapper = some text
bstrWrapper = Yet another string
bstrWrapper2 = some text
bstrWrapper = changing BSTR
bstrWrapper2 = some text
```

FINAL específico da Microsoft

Consulte também

[Classe _bstr_t](#)

_bstr_t::Attach

22/04/2020 • 2 minutes to read • [Edit Online](#)

Específico da Microsoft

Vincula um wrapper `_bstr_t` a um `BSTR`.

Sintaxe

```
void Attach(  
    BSTR s  
)
```

Parâmetros

`s`

Um `BSTR` a ser associado ou atribuído à variável `_bstr_t`.

Comentários

Se o `_bstr_t` estava associado anteriormente a outro `BSTR`, o `_bstr_t` limpará o recurso `BSTR`, se nenhuma outra variável `_bstr_t` estiver usando o `BSTR`.

Exemplo

Consulte [_bstr_t::Atribuir](#) para um exemplo usando **Anexar**.

Fim específico da Microsoft

Confira também

[Classe _bstr_t](#)

_bstr_t::_bstr_t

02/09/2020 • 2 minutes to read • [Edit Online](#)

Específico da Microsoft

Constrói um objeto `_bstr_t`.

Sintaxe

```
_bstr_t( ) throw( );
_bstr_t(
    const _bstr_t& s1
) throw( );
_bstr_t(
    const char* s2
);
_bstr_t(
    const wchar_t* s3
);
_bstr_t(
    const _variant_t& var
);
_bstr_t(
    BSTR bstr,
    bool fCopy
);
```

parâmetros

S1

Um objeto `_bstr_t` a ser copiado.

S2

Uma cadeia de caracteres multibyte.

Estado

Uma cadeia de caracteres Unicode

var

Um objeto `_variant_t`.

BSTR

Um objeto `BSTR` existente.

fCopy

Se `false`, o argumento *BSTR* será anexado ao novo objeto sem fazer uma cópia chamando `SysAllocString`.

Comentários

A tabela a seguir descreve os construtores `_bstr_t`.

CONSTRUTOR	DESCRIÇÃO
<code>_bstr_t()</code>	Constrói um objeto padrão <code>_bstr_t</code> que encapsula um <code>BSTR</code> objeto nulo.

CONSTRUTOR	DESCRIÇÃO
<code>_bstr_t(_bstr_t& s1)</code>	Constrói um objeto <code>_bstr_t</code> como uma cópia de outro. Essa é uma cópia <i>superficial</i> , que incrementa a contagem de referência do objeto encapsulado <code>BSTR</code> em vez de criar um novo.
<code>_bstr_t(char* s2)</code>	Constrói um objeto <code>_bstr_t</code> chamando <code>SysAllocString</code> para criar um novo objeto <code>BSTR</code> e encapsulá-lo. Esse construtor primeiro executa uma conversão de multibyte em Unicode.
<code>_bstr_t(wchar_t* s3)</code>	Constrói um objeto <code>_bstr_t</code> chamando <code>SysAllocString</code> para criar um novo objeto <code>BSTR</code> e encapsulá-lo.
<code>_bstr_t(_variant_t& var)</code>	Constrói um objeto <code>_bstr_t</code> de um objeto <code>_variant_t</code> recuperando primeiro um objeto <code>BSTR</code> do objeto VARIANT encapsulado.
<code>_bstr_t(BSTR bstr, bool fCopy)</code>	Constrói um objeto <code>_bstr_t</code> de um <code>BSTR</code> existente (em vez de uma cadeia de caracteres <code>wchar_t*</code>). Se <code>fCopy</code> for false, o fornecido <code>BSTR</code> será anexado ao novo objeto sem fazer uma nova cópia com <code>SysAllocString</code> . Esse construtor é usado por funções wrapper nos cabeçalhos da biblioteca de tipos para encapsular e assumir a propriedade de <code>BSTR</code> que é retornada por um método de interface.

FINAL específico da Microsoft

Confira também

[Classe _bstr_t](#)

[Classe _variant_t](#)

_bstr_t::copy

02/09/2020 • 2 minutes to read • [Edit Online](#)

Específico da Microsoft

Constrói uma cópia do `BSTR` encapsulado.

Sintaxe

```
BSTR copy( bool fCopy = true ) const;
```

parâmetros

fCopy

Se `true`, `Copy` retorna uma cópia do contido `BSTR`, caso contrário, `Copy` retorna o `BSTR` real.

Comentários

Retorna uma cópia recém-alocada do objeto `BSTR` encapsulado.

Exemplo

```
STDMETHODIMP CAlertMsg::get_ConnectionStr(BSTR *pVal){ //  m_bsConStr is _bstr_t
    *pVal = m_bsConStr.copy();
}
```

FINAL específico da Microsoft

Confira também

[Classe _bstr_t](#)

_bstr_t::Detach

25/03/2020 • 2 minutes to read • [Edit Online](#)

Seção específica da Microsoft

Retorna o `BSTR` encapsulado por um `_bstr_t` e desanexa o `BSTR` do `_bstr_t`.

Sintaxe

```
BSTR Detach( ) throw;
```

Valor retornado

O `BSTR` envolvido por `_bstr_t`.

Exemplo

Consulte [_bstr_t::assign](#) para obter um exemplo usando `Detach`.

Fim da seção específica da Microsoft

Confira também

[Classe _bstr_t](#)

_bstr_t::GetAddress

25/03/2020 • 2 minutes to read • [Edit Online](#)

Seção específica da Microsoft

Libera qualquer cadeia de caracteres existente e retorna o endereço de uma cadeia de caracteres recém-alocada.

Sintaxe

```
BSTR* GetAddress( );
```

Valor retornado

Um ponteiro para o `BSTR` encapsulado por `_bstr_t`.

Comentários

`GetAddress` afeta todos os objetos de `_bstr_t` que compartilham um `BSTR`. Mais de um `_bstr_t` pode compartilhar uma `BSTR` por meio do uso do construtor de cópia e do operador `=`.

Exemplo

Consulte [_bstr_t::assign](#) para obter um exemplo usando `GetAddress`.

Fim da seção específica da Microsoft

Confira também

[Classe _bstr_t](#)

_bstr_t::GetBSTR

25/03/2020 • 2 minutes to read • [Edit Online](#)

Seção específica da Microsoft

Aponta para o início do `BSTR` encapsulado por `_bstr_t`.

Sintaxe

```
BSTR& GetBSTR();
```

Valor retornado

O início do `BSTR` encapsulado por `_bstr_t`.

Comentários

`Getbstr` afeta todos os objetos de `_bstr_t` que compartilham um `BSTR`. Mais de um `_bstr_t` pode compartilhar uma `BSTR` por meio do uso do construtor de cópia e do operador `=`.

Exemplo

Consulte [_bstr_t: assign](#) para obter um exemplo usando `getbstr`.

Fim da seção específica da Microsoft

Confira também

[Classe _bstr_t](#)

_bstr_t::length

25/03/2020 • 2 minutes to read • [Edit Online](#)

Seção específica da Microsoft

Retorna o número de caracteres no `_bstr_t`, não incluindo a terminação nula, do `BSTR` encapsulado.

Sintaxe

```
unsigned int length( ) const throw( );
```

Comentários

Fim da seção específica da Microsoft

Confira também

[Classe _bstr_t](#)

Operadores (_bstr_t)

25/03/2020 • 2 minutes to read • [Edit Online](#)

Para obter informações sobre os operadores de `_bstr_t`, consulte [_Bstr_t classe](#).

Confira também

[Classe _bstr_t](#)

_bstr_t::operator =

25/03/2020 • 2 minutes to read • [Edit Online](#)

Seção específica da Microsoft

Atribui um novo valor a um objeto `_bstr_t` existente.

Sintaxe

```
_bstr_t& operator=(const _bstr_t& s1) throw ( );
_bstr_t& operator=(const char* s2);
_bstr_t& operator=(const wchar_t* s3);
_bstr_t& operator=(const _variant_t& var);
```

parâmetros

s1

Um objeto `_bstr_t` a ser atribuído a um objeto existente `_bstr_t`.

s2

Uma cadeia de caracteres multibyte a ser atribuída a um objeto `_bstr_t` existente.

Estado

Uma cadeia de caracteres Unicode a ser atribuída a um objeto `_bstr_t` existente.

var

Um objeto `_variant_t` a ser atribuído a um objeto existente `_bstr_t`.

Fim da seção específica da Microsoft

Exemplo

Consulte [_bstr_t::assign](#) para obter um exemplo de uso de Operator = .

Confira também

[Classe _bstr_t](#)

_bstr_t::operator +=, +

25/03/2020 • 2 minutes to read • [Edit Online](#)

Seção específica da Microsoft

Acrescenta caracteres ao final do objeto `_bstr_t` ou concatena duas cadeias de caracteres.

Sintaxe

```
_bstr_t& operator+=( const _bstr_t& s1 );
_bstr_t operator+( const _bstr_t& s1 );
friend _bstr_t operator+( const char* s2, const _bstr_t& s1);
friend _bstr_t operator+( const wchar_t* s3, const _bstr_t& s1);
```

parâmetros

s1

Um objeto `_bstr_t`.

s2

Uma cadeia de caracteres multibyte.

Estado

Uma cadeia de caracteres Unicode.

Comentários

Esses operadores executam a concatenação de cadeias de caracteres:

- **operador += (*S1*)** Acrescenta os caracteres no `BSTR` encapsulado de *S1* ao final do `BSTR` encapsulado desse objeto.
- **operador + (*S1*)** Retorna o novo `_bstr_t` que é formado concatenando o `BSTR` do objeto com o de *S1*.
- **operador + (*S2|S1*)** Retorna uma nova `_bstr_t` que é formada pela concatenação de uma cadeia de caracteres de vários bytes *S2*, convertida em Unicode, com o `BSTR` encapsulado em *S1*.
- **operador + (*S3, S1*)** Retorna uma nova `_bstr_t` que é formada pela concatenação de uma cadeia de caracteres Unicode *S3* com o `BSTR` encapsulado em *S1*.

Fim da seção específica da Microsoft

Confira também

[Classe _bstr_t](#)

_bstr_t::operator !

02/09/2020 • 2 minutes to read • [Edit Online](#)

Específico da Microsoft

Verifica se o encapsulado `BSTR` é uma cadeia de caracteres nula.

Sintaxe

```
bool operator!( ) const throw( );
```

Valor retornado

Ele retornará `true` se Sim, `false` se não.

FINAL específico da Microsoft

Confira também

[Classe _bstr_t](#)

Operadores relacionais `_bstr_t`

02/09/2020 • 2 minutes to read • [Edit Online](#)

Específico da Microsoft

Compara dois objetos `_bstr_t`.

Sintaxe

```
bool operator!( ) const throw( );
bool operator==(const _bstr_t& str) const throw( );
bool operator!=(const _bstr_t& str) const throw( );
bool operator<(const _bstr_t& str) const throw( );
bool operator>(const _bstr_t& str) const throw( );
bool operator<=(const _bstr_t& str) const throw( );
bool operator>=(const _bstr_t& str) const throw( );
```

Comentários

Esses operadores comparam dois objetos `_bstr_t` lexicograficamente. Os operadores retornam `true` se as comparações retêm; caso contrário, retornam `false`.

FINAL específico da Microsoft

Confira também

[Classe `_bstr_t`](#)

`_bstr_t::wchar_t*`, `_bstr_t::char*`

25/03/2020 • 2 minutes to read • [Edit Online](#)

Seção específica da Microsoft

Retorna os caracteres BSTR como uma matriz de caracteres estreitos ou largos.

Sintaxe

```
operator const wchar_t*( ) const throw( );
operator wchar_t*( ) const throw( );
operator const char*( ) const;
operator char*( ) const;
```

Comentários

Esses operadores podem ser usados para extrair os dados de caractere que são encapsuladas pelo objeto `BSTR`. Atribuir um novo valor para o ponteiro retornado não altera os dados originais de BSTR.

Fim da seção específica da Microsoft

Confira também

[Classe `_bstr_t`](#)

Classe _com_error

02/09/2020 • 2 minutes to read • [Edit Online](#)

Específico da Microsoft

Um objeto `_com_error` representa uma condição de exceção detectada pelas funções de wrapper de tratamento de erros nos arquivos de cabeçalho gerados da biblioteca de tipos ou por uma das classes de suporte com. A classe `_com_error` encapsula o código de erro HRESULT e qualquer `IErrorInfo` Interface objeto associado.

Construção

NOME	DESCRÍÇÃO
<code>_com_error</code>	Constrói um objeto <code>_com_error</code> .

Operadores

NOME	DESCRÍÇÃO
<code>operador =</code>	Atribui um objeto de <code>_com_error</code> existente para outro.

Funções de extrator

NOME	DESCRÍÇÃO
<code>Erro</code>	Recupera o HRESULT passado para o construtor.
<code>ErrorInfo</code>	Recupera o objeto <code>IErrorInfo</code> passado para o construtor.
<code>WCode</code>	Recupera o código de erro de 16 bits mapeado para o HRESULT encapsulado.

Funções IErrorInfo

NOME	DESCRÍÇÃO
<code>Descrição</code>	Chama a função <code>IErrorInfo::GetDescription</code> .
<code>Identificação</code>	Chama a função <code>IErrorInfo::GetHelpContext</code> .
<code>HelpFile</code>	Chama a função <code>IErrorInfo::GetHelpFile</code> .
<code>Origem</code>	Chama a função <code>IErrorInfo::GetSource</code> .
<code>GUID</code>	Chama a função <code>IErrorInfo::GetGUID</code> .

Extrator de mensagem de formato

NOME	DESCRIÇÃO
ErrorMessage	Recupera a mensagem de cadeia de caracteres para HRESULT armazenado no objeto <code>_com_error</code> .

Mapeadores de `ExepInfo.wCode` para `HRESULT`

NOME	DESCRIÇÃO
<code>HRESULTToWCode</code>	Mapeia 32 bits <code>HRESULT</code> para 16 bits <code>wCode</code> .
<code>WCodeToHRESULT</code>	Mapeia 16 bits <code>wCode</code> para <code>HRESULT</code> de 32 bits.

FINAL específico da Microsoft

Requisitos

Cabeçalho: <comdef.h>

Lib: `comsuppw.lib` ou `comsuppwd.lib` (consulte [/Zc: wchar_t \(Wchar_t é o tipo nativo\)](#) para obter mais informações)

Confira também

[Classes de suporte de COM do compilador](#)

[Interface IErrorInfo](#)

Funções de membro `_com_error`

25/03/2020 • 2 minutes to read • [Edit Online](#)

Para obter informações sobre as funções de membro `_com_error` , consulte [_com_error classe](#).

Confira também

[Classe _com_error](#)

_com_error::_com_error

25/03/2020 • 2 minutes to read • [Edit Online](#)

Seção específica da Microsoft

Constrói um objeto `_com_error`.

Sintaxe

```
_com_error(
    HRESULT hr,
    IErrorInfo* perrinfo = NULL,
    bool fAddRef=false) throw( );

_com_error( const _com_error& that ) throw( );
```

parâmetros

Human

Informações de HRESULT.

perrinfo

Objeto `IErrorInfo`.

fAddRef

O padrão faz com que o Construtor chame AddRef em uma interface de `IErrorInfo` não nula. Isso fornece a contagem de referência correta no caso comum em que a propriedade da interface é passada para o objeto `_com_error`, como:

```
throw _com_error(hr, perrinfo);
```

Se você não quiser que seu código transfira a propriedade para o objeto `_com_error`, e a `AddRef` for necessária para compensar o `Release` no destruidor `_com_error`, construa o objeto da seguinte maneira:

```
_com_error err(hr, perrinfo, true);
```

Isto

Um objeto `_com_error` existente.

Comentários

O primeiro construtor cria um novo objeto, dado um HRESULT e um objeto de `IErrorInfo` opcional. O segundo cria uma cópia de um objeto de `_com_error` existente.

Fim da seção específica da Microsoft

Confira também

[Classe _com_error](#)

_com_error::Description

25/03/2020 • 2 minutes to read • [Edit Online](#)

Seção específica da Microsoft

Chama a função `IErrorInfo::GetDescription`.

Sintaxe

```
_bstr_t Description( ) const;
```

Valor retornado

Retorna o resultado de `IErrorInfo::GetDescription` para o objeto de `IErrorInfo` registrado no objeto `_com_error`. O `BSTR` resultante é encapsulado em um objeto `_bstr_t`. Se nenhum `IErrorInfo` for registrado, ele retornará um `_bstr_t` vazio.

Comentários

Chama a função `IErrorInfo::GetDescription` e recupera `IErrorInfo` registradas no objeto `_com_error`. Qualquer falha ao chamar o método `IErrorInfo::GetDescription` é ignorada.

Fim da seção específica da Microsoft

Confira também

[Classe _com_error](#)

_com_error::Error

25/03/2020 • 2 minutes to read • [Edit Online](#)

Seção específica da Microsoft

Recupera o HRESULT passado para o construtor.

Sintaxe

```
HRESULT Error( ) const throw( );
```

Valor retornado

Item HRESULT bruto passado para o construtor.

Comentários

Recupera o item de HRESULT encapsulado em um objeto `_com_error`.

Fim da seção específica da Microsoft

Confira também

[Classe _com_error](#)

_com_error::ErrorInfo

25/03/2020 • 2 minutes to read • [Edit Online](#)

Seção específica da Microsoft

Recupera o objeto `IErrorInfo` passado para o construtor.

Sintaxe

```
IErrorInfo * ErrorInfo( ) const throw( );
```

Valor retornado

O item `IErrorInfo` bruto passado para o construtor.

Comentários

Recupera o item de `IErrorInfo` encapsulado em um objeto `_com_error` ou nulo se nenhum item de `IErrorInfo` for registrado. O chamador deve chamar `Release` no objeto retornado quando terminar de usá-lo.

Fim da seção específica da Microsoft

Confira também

[Classe _com_error](#)

_com_error::ErrorMessage

25/03/2020 • 2 minutes to read • [Edit Online](#)

Seção específica da Microsoft

Recupera a mensagem de cadeia de caracteres para o HRESULT armazenado no objeto `_com_error`.

Sintaxe

```
const TCHAR * ErrorMessage( ) const throw( );
```

Valor retornado

Retorna a mensagem de cadeia de caracteres para o HRESULT registrado no objeto `_com_error`. Se o HRESULT for um `wCode` de 16 bits mapeado, uma mensagem genérica " `IDispatch error #<wCode>` " será retornada. Se nenhuma mensagem for encontrada, então uma mensagem genérica " `Unknown error #<HRESULT>` " será retornada. A cadeia de caracteres retornada será uma cadeia de caracteres Unicode ou multibyte, dependendo do estado da macro `_UNICODE`.

Comentários

Recupera o texto apropriado da mensagem do sistema para HRESULT registrado no objeto `_com_error`. O texto da mensagem do sistema é obtido chamando a função `FormatMessage` do Win32. A cadeia de caracteres retornada é alocada pela API `FormatMessage` e é liberada quando o objeto `_com_error` é destruído.

Fim da seção específica da Microsoft

Confira também

[Classe _com_error](#)

_com_error::GUID

25/03/2020 • 2 minutes to read • [Edit Online](#)

Seção específica da Microsoft

Chama a função `IErrorInfo::GetGUID`.

Sintaxe

```
GUID GUID( ) const throw( );
```

Valor retornado

Retorna o resultado de `IErrorInfo::GetGUID` para o objeto de `IErrorInfo` registrado no objeto `_com_error`. Se nenhum objeto de `IErrorInfo` for registrado, ele retornará `GUID_NULL`.

Comentários

Qualquer falha ao chamar o método `IErrorInfo::GetGUID` é ignorada.

Fim da seção específica da Microsoft

Confira também

[Classe _com_error](#)

_com_error::HelpContext

25/03/2020 • 2 minutes to read • [Edit Online](#)

Seção específica da Microsoft

Chama a função `IErrorInfo::GetHelpContext`.

Sintaxe

```
DWORD HelpContext( ) const throw( );
```

Valor retornado

Retorna o resultado de `IErrorInfo::GetHelpContext` para o objeto de `IErrorInfo` registrado no objeto `_com_error`. Se nenhum objeto de `IErrorInfo` for registrado, ele retornará um zero.

Comentários

Qualquer falha ao chamar o método `IErrorInfo::GetHelpContext` é ignorada.

Fim da seção específica da Microsoft

Confira também

[Classe _com_error](#)

_com_error::HelpFile

25/03/2020 • 2 minutes to read • [Edit Online](#)

Seção específica da Microsoft

Chama a função `IErrorInfo::GetHelpFile`.

Sintaxe

```
_bstr_t HelpFile() const;
```

Valor retornado

Retorna o resultado de `IErrorInfo::GetHelpFile` para o objeto de `IErrorInfo` registrado no objeto `_com_error`. O BSTR resultante é encapsulado em um objeto `_bstr_t`. Se nenhum `IErrorInfo` for registrado, ele retornará um `_bstr_t` vazio.

Comentários

Qualquer falha ao chamar o método `IErrorInfo::GetHelpFile` é ignorada.

Fim da seção específica da Microsoft

Confira também

[Classe _com_error](#)

_com_error::HRESULTToWCode

25/03/2020 • 2 minutes to read • [Edit Online](#)

Seção específica da Microsoft

Mapeia 32 bits de HRESULT para `wCode` de 16 bits.

Sintaxe

```
static WORD HRESULTToWCode(  
    HRESULT hr  
) throw( );
```

parâmetros

Human

O HRESULT de 32 bits a ser mapeado para `wCode` de 16 bits.

Valor retornado

`wCode` de 16 bits mapeada do HRESULT de 32 bits.

Comentários

Consulte [_com_error:: WCode](#) para obter mais informações.

Fim da seção específica da Microsoft

Confira também

[_com_error::WCode](#)

[_com_error::WCodeToHRESULT](#)

[Classe _com_error](#)

_com_error::Source

25/03/2020 • 2 minutes to read • [Edit Online](#)

Seção específica da Microsoft

Chama a função `IErrorInfo::GetSource`.

Sintaxe

```
_bstr_t Source() const;
```

Valor retornado

Retorna o resultado de `IErrorInfo::GetSource` para o objeto de `IErrorInfo` registrado no objeto `_com_error`. O `BSTR` resultante é encapsulado em um objeto `_bstr_t`. Se nenhum `IErrorInfo` for registrado, ele retornará um `_bstr_t` vazio.

Comentários

Qualquer falha ao chamar o método `IErrorInfo::GetSource` é ignorada.

Fim da seção específica da Microsoft

Confira também

[Classe _com_error](#)

_com_error::WCode

25/03/2020 • 2 minutes to read • [Edit Online](#)

Seção específica da Microsoft

Recupera o código de erro de 16 bits mapeado para o HRESULT encapsulado.

Sintaxe

```
WORD WCode( ) const throw( );
```

Valor retornado

Se o HRESULT estiver dentro do intervalo de 0x80040200 a 0x8004FFFF, o método `wcode` retornará o HRESULT menos 0x80040200; caso contrário, retornará zero.

Comentários

O método `WCode` é usado para desfazer um mapeamento que ocorre no código de suporte COM. O wrapper de uma `dispinterface` propriedade ou método chama uma rotina de suporte que empacota os argumentos e chamadas `IDispatch::Invoke`. No retorno, se um HRESULT de falha de `DISP_E_EXCEPTION` for retornado, as informações de erro serão recuperadas da estrutura de `EXCEPINFO` passada para `IDispatch::Invoke`. O código de erro pode ser um valor de 16 bits armazenado no membro de `wcode` da estrutura de `EXCEPINFO` ou um valor de 32 bits completo no membro `scode` da estrutura de `EXCEPINFO`. Se um `wCode` de 16 bits for retornado, ele deverá primeiro ser mapeado para um HRESULT de falha de 32 bits.

[Fim da seção específica da Microsoft](#)

Confira também

[_com_error::HRESULTToWCode](#)

[_com_error::WCodeToHRESULT](#)

[Classe _com_error](#)

_com_error::WCodeToHRESULT

25/03/2020 • 2 minutes to read • [Edit Online](#)

Seção específica da Microsoft

Mapeia *wCode* de 16 bits para HRESULT de 32 bits.

Sintaxe

```
static HRESULT WCodeToHRESULT(
    WORD wCode
) throw( );
```

parâmetros

wCode

O *wCode* de 16 bits a ser MAPEADO para HRESULT de 32 bits.

Valor retornado

HRESULT de 32 bits mapeado do *wCode* de 16 bits.

Comentários

Consulte a função membro [WCode](#).

Fim da seção específica da Microsoft

Confira também

[_com_error::WCode](#)

[_com_error::HRESULTToWCode](#)

[Classe _com_error](#)

Operadores (_com_error)

25/03/2020 • 2 minutes to read • [Edit Online](#)

Para obter informações sobre os operadores de `_com_error` , consulte [_com_error classe](#).

Confira também

[Classe _com_error](#)

_com_error::operator =

25/03/2020 • 2 minutes to read • [Edit Online](#)

Seção específica da Microsoft

Atribui um objeto `_com_error` existente a outro.

Sintaxe

```
_com_error& operator = (
    const _com_error& that
) throw ( );
```

parâmetros

Isso

Um objeto `_com_error`.

Fim da seção específica da Microsoft

Confira também

[Classe _com_error](#)

Classe _com_ptr_t

02/09/2020 • 2 minutes to read • [Edit Online](#)

Específico da Microsoft

Um objeto `_com_ptr_t` encapsula um ponteiro de interface com e é chamado de ponteiro "inteligente". Essa classe de modelo gerencia a alocação de recursos e a desalocação por meio de chamadas de função para as `IUnknown` funções de membro: `QueryInterface` , `AddRef` e `Release` .

Um ponteiro inteligente é geralmente referenciado pela definição `typedef` fornecida pela macro `_COM_SMARTPTR_TYPEDEF`. Essa macro usa um nome de interface e o IID e declara uma especialização de `_com_ptr_t` com o nome da interface mais um sufixo de `Ptr` . Por exemplo:

```
_COM_SMARTPTR_TYPEDEF(IMyInterface, __uuidof(IMyInterface));
```

declara a especialização de `_com_ptr_t` `IMyInterfacePtr` .

Um conjunto de [modelos de função](#), não membros dessa classe de modelo, dá suporte a comparações com um ponteiro inteligente no lado direito do operador de comparação.

Construção

NOME	DESCRIÇÃO
<code>_com_ptr_t</code>	Constrói um objeto <code>_com_ptr_t</code> .

Operações de nível baixo

NOME	DESCRIÇÃO
<code>AddRef</code>	Chama a <code>AddRef</code> função de membro de <code>IUnknown</code> no ponteiro de interface encapsulada.
<code>Attach</code>	Encapsula um ponteiro de interface bruto desse tipo de ponteiro inteligente.
<code>CreateInstance</code>	Cria uma nova instância de um objeto dado a um <code>CLSID</code> ou <code>ProgID</code> .
<code>Desanexar</code>	Extrai e retorna o ponteiro de interface encapsulado.
<code>GetActiveObject</code>	Anexa a uma instância existente de um objeto, dado a <code>CLSID</code> ou <code>ProgID</code> .
<code>GetInterfacePtr</code>	Retorna o ponteiro de interface encapsulado.
<code>QueryInterface</code>	Chama a <code>QueryInterface</code> função de membro de <code>IUnknown</code> no ponteiro de interface encapsulada.
<code>Versão</code>	Chama a <code>Release</code> função de membro de <code>IUnknown</code> no ponteiro de interface encapsulada.

Operadores

NOME	DESCRIÇÃO
<code>operador =</code>	Atribui um novo valor a um objeto de <code>_com_ptr_t</code> existente.
<code>operadores =,! =, <, >, <=, >=</code>	Comparam o objeto de ponteiro inteligente com outro ponteiro inteligente, um ponteiro de interface bruto ou um NULL.
<code>Extratores</code>	Extrai o ponteiro de interface COM encapsulado.

FINAL específico da Microsoft

Requisitos

Cabeçalho: <comip.h>

Lib: comsuppw.lib ou comsuppwd.lib (consulte [/Zc: Wchar_t \(Wchar_t é o tipo nativo\)](#) para obter mais informações)

Confira também

[Classes de suporte de COM do compilador](#)

Funções de membro _com_ptr_t

25/03/2020 • 2 minutes to read • [Edit Online](#)

Para obter informações sobre as funções de membro `_com_ptr_t`, consulte [_com_ptr_t classe](#).

Confira também

[Classe _com_ptr_t](#)

_com_ptr_t::_com_ptr_t

02/09/2020 • 3 minutes to read • [Edit Online](#)

Específico da Microsoft

Constrói um objeto `_com_ptr_t`.

Sintaxe

```
// Default constructor.
// Constructs a NULL smart pointer.
_com_ptr_t() throw();

// Constructs a NULL smart pointer. The NULL argument must be zero.
_com_ptr_t(
    int null
);

// Constructs a smart pointer as a copy of another instance of the
// same smart pointer. AddRef is called to increment the reference
// count for the encapsulated interface pointer.
_com_ptr_t(
    const _com_ptr_t& cp
) throw();

// Move constructor (Visual Studio 2015 Update 3 and later)
_com_ptr_t(_com_ptr_t& cp) throw();

// Constructs a smart pointer from a raw interface pointer of this
// smart pointer's type. If fAddRef is true, AddRef is called
// to increment the reference count for the encapsulated
// interface pointer. If fAddRef is false, this constructor
// takes ownership of the raw interface pointer without calling AddRef.
_com_ptr_t(
    Interface* pInterface,
    bool fAddRef
) throw();

// Construct pointer for a _variant_t object.
// Constructs a smart pointer from a _variant_t object. The
// encapsulated VARIANT must be of type VT_DISPATCH or VT_UNKNOWN, or
// it can be converted into one of these two types. If QueryInterface
// fails with an E_NOINTERFACE error, a NULL smart pointer is
// constructed.
_com_ptr_t(
    const _variant_t& varSrc
);

// Constructs a smart pointer given the CLSID of a coclass. This
// function calls CoCreateInstance, by the member function
// CreateInstance, to create a new COM object and then queries for
// this smart pointer's interface type. If QueryInterface fails with
// an E_NOINTERFACE error, a NULL smart pointer is constructed.
explicit _com_ptr_t(
    const CLSID& clsid,
    IUnknown* pOuter = NULL,
    DWORD dwClsContext = CLSCTX_ALL
);

// Calls CoCreateClass with provided CLSID retrieved from string.
explicit _com_ptr_t(
```

```

LPCWSTR str,
IUnknown* pOuter = NULL,
DWORD dwClsContext = CLSCTX_ALL
);

// Constructs a smart pointer given a multibyte character string that
// holds either a CLSID (starting with "{") or a ProgID. This function
// calls CoCreateInstance, by the member function CreateInstance, to
// create a new COM object and then queries for this smart pointer's
// interface type. If QueryInterface fails with an E_NOINTERFACE error,
// a NULL smart pointer is constructed.
explicit _com_ptr_t(
    LPCSTR str,
    IUnknown* pOuter = NULL,
    DWORD dwClsContext = CLSCTX_ALL
);

// Saves the interface.
template<>
_com_ptr_t(
    Interface* pInterface
) throw();

// Make sure correct ctor is called
template<>
_com_ptr_t(
    LPSTR str
);

// Make sure correct ctor is called
template<>
_com_ptr_t(
    LPWSTR str
);

// Constructs a smart pointer from a different smart pointer type or
// from a different raw interface pointer. QueryInterface is called to
// find an interface pointer of this smart pointer's type. If
// QueryInterface fails with an E_NOINTERFACE error, a NULL smart
// pointer is constructed.
template<typename _OtherIID>
_com_ptr_t(
    const _com_ptr_t<_OtherIID>& p
);

// Constructs a smart-pointer from any IUnknown-based interface pointer.
template<typename _InterfaceType>
_com_ptr_t(
    _InterfaceType* p
);

// Disable conversion using _com_ptr_t* specialization of
// template<typename _InterfaceType> _com_ptr_t(_InterfaceType* p)
template<>
explicit _com_ptr_t(
    _com_ptr_t* p
);

```

parâmetros

pInterface

Um ponteiro de interface bruto.

fAddRef

Se `true` , `AddRef` é chamado para incrementar a contagem de referência do ponteiro de interface encapsulado.

CP

Um objeto `_com_ptr_t`.

DTI

Um ponteiro de interface bruto, seu tipo sendo diferente do tipo de ponteiro inteligente deste `_com_ptr_t` objeto.

`{1>varSrc<1}`

Um objeto `_variant_t`.

clsid

O `CLSID` de uma coclass.

dwClsContext

Contexto para execução do código executável.

lpcStr

Uma cadeia de caracteres multibyte que contém um `CLSID` (começando com "{") ou um `ProgID`.

pOuter

O desconhecido externo para [agregação](#).

FINAL específico da Microsoft

Confira também

[Classe `_com_ptr_t`](#)

_com_ptr_t::AddRef

22/04/2020 • 2 minutes to read • [Edit Online](#)

Específico da Microsoft

Chama `AddRef` a função `IUnknown` de membro do ponteiro de interface encapsulado.

Sintaxe

```
void AddRef( );
```

Comentários

Chama `IUnknown::AddRef` o ponteiro de interface `E_POINTER` encapsulado, levantando um erro se o ponteiro for NULO.

Fim específico da Microsoft

Confira também

[Classe _com_ptr_t](#)

_com_ptr_t::Attach

02/09/2020 • 2 minutes to read • [Edit Online](#)

Específico da Microsoft

Encapsula um ponteiro de interface bruto desse tipo de ponteiro inteligente.

Sintaxe

```
void Attach( Interface* pInterface ) throw( );
void Attach( Interface* pInterface, bool fAddRef ) throw( );
```

parâmetros

pInterface

Um ponteiro de interface bruto.

fAddRef

Se for `true`, `AddRef` será chamado. Se for `false`, o `_com_ptr_t` objeto assumirá a propriedade do ponteiro de interface bruto sem chamar `AddRef`.

Comentários

- **Anexar (*pInterface*)** `AddRef` Não é chamado. A propriedade da interface é transmitida a este objeto `_com_ptr_t`. `Release` é chamado para diminuir a contagem de referência para o ponteiro encapsulado anteriormente.
- **Attach (*pInterface*, *fAddRef*)** Se *fAddRef* for `true`, `AddRef` será chamado para incrementar a contagem de referência para o ponteiro de interface encapsulada. Se *fAddRef* for `false`, esse `_com_ptr_t` objeto assumirá a propriedade do ponteiro de interface bruto sem chamar `AddRef`. `Release` é chamado para diminuir a contagem de referência para o ponteiro encapsulado anteriormente.

FINAL específico da Microsoft

Confira também

[Classe _com_ptr_t](#)

_com_ptr_t::CreateInstance

25/03/2020 • 2 minutes to read • [Edit Online](#)

Seção específica da Microsoft

Cria uma nova instância de um objeto, de acordo com um `CLSID` ou `ProgID`.

Sintaxe

```
HRESULT CreateInstance(
    const CLSID& rclsid,
    IUnknown* pOuter=NULL,
    DWORD dwClsContext = CLSCTX_ALL
) throw( );
HRESULT CreateInstance(
    LPCWSTR clsidString,
    IUnknown* pOuter=NULL,
    DWORD dwClsContext = CLSCTX_ALL
) throw( );
HRESULT CreateInstance(
    LPCSTR clsidStringA,
    IUnknown* pOuter=NULL,
    DWORD dwClsContext = CLSCTX_ALL
) throw( );
```

parâmetros

rclsid

A `CLSID` de um objeto.

clsidString

Uma cadeia de caracteres Unicode que mantém um `CLSID` (começando com "{ ") ou um `ProgID`.

clsidStringA

Uma cadeia de caracteres multibyte, usando a página de código ANSI, que contém um `CLSID` (começando com "{ ") ou um `ProgID`.

dwClsContext

Contexto para execução do código executável.

pOuter

O desconhecido externo para [agregação](#).

Comentários

Essas funções membro chamam `CoCreateInstance` para criar um novo objeto COM e, em seguida, buscam o tipo de interface desse ponteiro inteligente. O ponteiro resultante é encapsulado nesse objeto `_com_ptr_t`. `Release` é chamado para diminuir a contagem de referência para o ponteiro encapsulado anteriormente. Essa rotina retorna o HRESULT para indicar êxito ou falha.

- **CreateInstance (*rclsid*, *dwClsContext*)** Cria uma nova instância em execução de um objeto, dado um `CLSID`.
- **CreateInstance (*clsidString*, *dwClsContext*)** Cria uma nova instância em execução de um objeto, considerando uma cadeia de caracteres Unicode que contém um `CLSID` (começando com "{ ") ou um

ProgID .

- **CreateInstance (*clsidStringA*, *dwClsContext*)** Cria uma nova instância em execução de um objeto, dado uma cadeia de caracteres multibyte que contém um **CLSID** (começando com " { ") ou um **ProgID** . Chama [MultiByteToWideChar](#), que pressupõe que a cadeia de caracteres está na página de código ANSI em vez de uma página de código OEM.

Fim da seção específica da Microsoft

Confira também

[Classe _com_ptr_t](#)

_com_ptr_t::Detach

25/03/2020 • 2 minutes to read • [Edit Online](#)

Seção específica da Microsoft

Extrai e retorna o ponteiro de interface encapsulado.

Sintaxe

```
Interface* Detach( ) throw( );
```

Comentários

Extrai e retorna o ponteiro de interface encapsulado, e depois limpa o armazenamento de ponteiro encapsulado para NULL. Isso remove o ponteiro de interface do encapsulamento. Cabe a você chamar `Release` no ponteiro de interface retornado.

Fim da seção específica da Microsoft

Confira também

[Classe _com_ptr_t](#)

_com_ptr_t::GetActiveObject

25/03/2020 • 2 minutes to read • [Edit Online](#)

Seção específica da Microsoft

Anexa a uma instância existente de um objeto, de acordo com uma `CLSID` ou `ProgID`.

Sintaxe

```
HRESULT GetActiveObject(
    const CLSID& rclsid
) throw( );
HRESULT GetActiveObject(
    LPCWSTR clsidString
) throw( );
HRESULT GetActiveObject(
    LPCSTR clsidStringA
) throw( );
```

parâmetros

rclsid

A `CLSID` de um objeto.

clsidString

Uma cadeia de caracteres Unicode que mantém um `CLSID` (começando com "{ ") ou um `ProgID`.

clsidStringA

Uma cadeia de caracteres multibyte, usando a página de código ANSI, que contém um `CLSID` (começando com "{ ") ou um `ProgID`.

Comentários

Essas funções de membro chamam `GetActiveObject` para recuperar um ponteiro para um objeto em execução que foi registrado com OLE e, em seguida, consulta o tipo de interface do ponteiro inteligente. O ponteiro resultante é encapsulado nesse objeto `_com_ptr_t`. `Release` é chamado para diminuir a contagem de referência para o ponteiro encapsulado anteriormente. Essa rotina retorna o `HRESULT` para indicar êxito ou falha.

- `GetActiveObject (rclsid)` Anexa a uma instância existente de um objeto, dado um `CLSID`.
- `GetActiveObject (clsidString)` Anexa a uma instância existente de um objeto, dada uma cadeia de caracteres Unicode que mantém um `CLSID` (começando com "{ ") ou um `ProgID`.
- `GetActiveObject (clsidStringA)` Anexa a uma instância existente de um objeto, dado uma cadeia de caracteres multibyte que contém um `CLSID` (começando com "{ ") ou um `ProgID`. Chama `MultiByteToWideChar`, que pressupõe que a cadeia de caracteres está na página de código ANSI em vez de uma página de código OEM.

Fim da seção específica da Microsoft

Confira também

[Classe _com_ptr_t](#)

_com_ptr_t::GetInterfacePtr

25/03/2020 • 2 minutes to read • [Edit Online](#)

Seção específica da Microsoft

Retorna o ponteiro de interface encapsulado.

Sintaxe

```
Interface* GetInterfacePtr( ) const throw( );
Interface*& GetInterfacePtr() throw();
```

Comentários

Retorna o ponteiro de interface encapsulada, que pode ser NULL.

Fim da seção específica da Microsoft

Confira também

[Classe _com_ptr_t](#)

_com_ptr_t::QueryInterface

25/03/2020 • 2 minutes to read • [Edit Online](#)

Seção específica da Microsoft

Chama a função de membro `QueryInterface` de `IUnknown` no ponteiro de interface encapsulada.

Sintaxe

```
template<typename _InterfaceType> HRESULT QueryInterface (  
    const IID& iid,  
    _InterfaceType*& p  
) throw ( );  
template<typename _InterfaceType> HRESULT QueryInterface (   
    const IID& iid,  
    _InterfaceType** p  
) throw( );
```

parâmetros

`IID`

`IUnknown` de um ponteiro de interface.

`p`

Ponteiro da interface bruta.

Comentários

Chama `IUnknown::QueryInterface` no ponteiro de interface encapsulado com o `IID` especificado e retorna o ponteiro de interface bruta resultante em `p`. Essa rotina retorna o HRESULT para indicar êxito ou falha.

Fim da seção específica da Microsoft

Confira também

[Classe _com_ptr_t](#)

_com_ptr_t::Release

22/04/2020 • 2 minutes to read • [Edit Online](#)

Específico da Microsoft

Chama a função `IUnknown` de membro de **versão** do ponteiro de interface encapsulado.

Sintaxe

```
void Release( );
```

Comentários

Chama `IUnknown::Release` o ponteiro de interface `E_POINTER` encapsulado, levantando um erro se este ponteiro de interface for NULL.

Fim específico da Microsoft

Confira também

[Classe _com_ptr_t](#)

Operadores (_com_ptr_t)

25/03/2020 • 2 minutes to read • [Edit Online](#)

Para obter informações sobre os operadores de `_com_ptr_t`, consulte [_Com_ptr_t classe](#).

Confira também

[Classe _com_ptr_t](#)

_com_ptr_t::operator =

25/03/2020 • 2 minutes to read • [Edit Online](#)

Seção específica da Microsoft

Atribui um novo valor a um objeto `_com_ptr_t` existente.

Sintaxe

```
template<typename _OtherIID>
_com_ptr_t& operator=( const _com_ptr_t<_OtherIID>& p );

// Sets a smart pointer to be a different smart pointer of a different
// type or a different raw interface pointer. QueryInterface is called
// to find an interface pointer of this smart pointer's type, and
// Release is called to decrement the reference count for the previously
// encapsulated pointer. If QueryInterface fails with an E_NOINTERFACE,
// a NULL smart pointer results.
template<typename _InterfaceType>
_com_ptr_t& operator=( _InterfaceType* p );

// Encapsulates a raw interface pointer of this smart pointer's type.
// AddRef is called to increment the reference count for the encapsulated
// interface pointer, and Release is called to decrement the reference
// count for the previously encapsulated pointer.
template<> _com_ptr_t&
operator=( Interface* pInterface ) throw();

// Sets a smart pointer to be a copy of another instance of the same
// smart pointer of the same type. AddRef is called to increment the
// reference count for the encapsulated interface pointer, and Release
// is called to decrement the reference count for the previously
// encapsulated pointer.
_com_ptr_t& operator=( const _com_ptr_t& cp ) throw();

// Sets a smart pointer to NULL. The NULL argument must be a zero.
_com_ptr_t& operator=( int null );

// Sets a smart pointer to be a _variant_t object. The encapsulated
// VARIANT must be of type VT_DISPATCH or VT_UNKNOWN, or it can be
// converted to one of these two types. If QueryInterface fails with an
// E_NOINTERFACE error, a NULL smart pointer results.
_com_ptr_t& operator=( const _variant_t& varSrc );
```

Comentários

Atribui um ponteiro de interface para esse objeto `_com_ptr_t`.

Fim da seção específica da Microsoft

Confira também

[Classe _com_ptr_t](#)

Operadores relacionais _com_ptr_t

25/03/2020 • 2 minutes to read • [Edit Online](#)

Seção específica da Microsoft

Comparam o objeto de ponteiro inteligente com outro ponteiro inteligente, um ponteiro de interface bruto ou um NULL.

Sintaxe

```
template<typename _OtherIID>
bool operator==( const _com_ptr_t<_OtherIID>& p );

template<typename _OtherIID>
bool operator==( _com_ptr_t<_OtherIID>& p );

template<typename _InterfaceType>
bool operator==( _InterfaceType* p );

template<>
bool operator==( Interface* p );

template<>
bool operator==( const _com_ptr_t& p ) throw();

template<>
bool operator==( _com_ptr_t& p ) throw();

bool operator==( Int null );

template<typename _OtherIID>
bool operator!=( const _com_ptr_t<_OtherIID>& p );

template<typename _OtherIID>
bool operator!=( _com_ptr_t<_OtherIID>& p );

template<typename _InterfaceType>
bool operator!=( _InterfaceType* p );

bool operator!=( Int null );

template<typename _OtherIID>
bool operator<( const _com_ptr_t<_OtherIID>& p );

template<typename _OtherIID>
bool operator<( _com_ptr_t<_OtherIID>& p );

template<typename _InterfaceType>
bool operator<( _InterfaceType* p );

template<typename _OtherIID>
bool operator>( const _com_ptr_t<_OtherIID>& p );

template<typename _OtherIID>
bool operator>( _com_ptr_t<_OtherIID>& p );

template<typename _InterfaceType>
bool operator>( _InterfaceType* p );

template<typename _OtherIID>
bool operator<=( const _com_ptr_t<_OtherIID>& p );

template<typename _OtherIID>
bool operator<=( _com_ptr_t<_OtherIID>& p );

template<typename _InterfaceType>
bool operator<=( _InterfaceType* p );

template<typename _OtherIID>
bool operator>=( const _com_ptr_t<_OtherIID>& p );

template<typename _OtherIID>
bool operator>=( _com_ptr_t<_OtherIID>& p );

template<typename _InterfaceType>
bool operator>=( _InterfaceType* p );
```

Comentários

Compara um objeto ponteiro inteligente com outro ponteiro inteligente, um ponteiro de interface bruto ou um NULL. Exceto para os testes de ponteiro NULL, esses operadores primeiro consultam ambos os ponteiros para `IUnknown` e comparam os resultados.

Fim da seção específica da Microsoft

Confira também

[Classe _com_ptr_t](#)

`_com_ptr_t` Extratores

02/09/2020 • 2 minutes to read • [Edit Online](#)

Específico da Microsoft

Extrai o ponteiro de interface COM encapsulado.

Sintaxe

```
operator Interface*( ) const throw( );
operator Interface&( ) const;
Interface& operator*( ) const;
Interface* operator->( ) const;
Interface** operator&( ) throw( );
operator bool( ) const throw( );
```

Comentários

- `operator Interface*` Retorna o ponteiro de interface encapsulado, que pode ser nulo.
- `operator Interface&` Retorna uma referência ao ponteiro de interface encapsulado e emite um erro se o ponteiro for nulo.
- `operator*` Permite que um objeto de ponteiro inteligente atue como se fosse a interface encapsulada real quando não referenciada.
- `operator->` Permite que um objeto de ponteiro inteligente atue como se fosse a interface encapsulada real quando não referenciada.
- `operator&` Libera qualquer ponteiro de interface encapsulada, substituindo-o por NULL e retorna o endereço do ponteiro encapsulado. Esse operador permite passar o ponteiro inteligente por endereço para uma função que tem um parâmetro *out* por meio do qual ele retorna um ponteiro de interface.
- `operator bool` Permite que um objeto de ponteiro inteligente seja usado em uma expressão condicional. Esse operador retorna `true` se o ponteiro não for nulo.

NOTE

Como o `operator bool` não é declarado como `explicit`, `_com_ptr_t` é implicitamente conversível para `bool`, que é conversível para qualquer tipo escalar. Isso pode ter consequências inesperadas em seu código. Habilite o [aviso do compilador \(nível 4\) C4800](#) para impedir o uso não intencional dessa conversão.

Confira também

[Classe `_com_ptr_t`](#)

Modelos de função relacional

25/03/2020 • 2 minutes to read • [Edit Online](#)

Seção específica da Microsoft

Sintaxe

```

template<typename _InterfaceType> bool operator==(

    int NULL,
    _com_ptr_t<_InterfaceType>& p
);

template<typename _Interface,
         typename _InterfacePtr> bool operator==(

    _Interface* i,
    _com_ptr_t<_InterfacePtr>& p
);

template<typename _Interface> bool operator!=(

    int NULL,
    _com_ptr_t<_Interface>& p
);

template<typename _Interface,
         typename _InterfacePtr> bool operator!=(

    _Interface* i,
    _com_ptr_t<_InterfacePtr>& p
);

template<typename _Interface> bool operator<(

    int NULL,
    _com_ptr_t<_Interface>& p
);

template<typename _Interface,
         typename _InterfacePtr> bool operator<(

    _Interface* i,
    _com_ptr_t<_InterfacePtr>& p
);

template<typename _Interface> bool operator>(

    int NULL,
    _com_ptr_t<_Interface>& p
);

template<typename _Interface,
         typename _InterfacePtr> bool operator>(

    _Interface* i,
    _com_ptr_t<_InterfacePtr>& p
);

template<typename _Interface> bool operator<=(

    int NULL,
    _com_ptr_t<_Interface>& p
);

template<typename _Interface,
         typename _InterfacePtr> bool operator<=(

    _Interface* i,
    _com_ptr_t<_InterfacePtr>& p
);

template<typename _Interface> bool operator>=(

    int NULL,
    _com_ptr_t<_Interface>& p
);

template<typename _Interface,
         typename _InterfacePtr> bool operator>=(

    _Interface* i,
    _com_ptr_t<_InterfacePtr>& p
);

```

parâmetros

i

Um ponteiro de interface bruto.

p

Um ponteiro inteligente.

Comentários

Esses modelos de função permitem a comparação com um ponteiro inteligente à direita do operador de comparação. Eles não são funções de membro de `_com_ptr_t`.

Fim da seção específica da Microsoft

Confira também

[Classe _com_ptr_t](#)

Classe _variant_t

02/09/2020 • 2 minutes to read • [Edit Online](#)

Específico da Microsoft

Um objeto _variant_t encapsula o `VARIANT` tipo de dados. A classe gerencia a alocação de recursos e a desalocação e faz chamadas de função para `VariantInit` e `VariantClear` conforme apropriado.

Construção

NOME	DESCRIÇÃO
<code>_variant_t</code>	Constrói um objeto _variant_t .

Operations

NOME	DESCRIÇÃO
<code>Attach</code>	Anexa um <code>VARIANT</code> objeto ao objeto _variant_t .
<code>Limpar</code>	Limpa o objeto encapsulado <code>VARIANT</code> .
<code>ChangeType</code>	Altera o tipo do objeto _variant_t para o indicado <code>VARTYPE</code> .
<code>Desanexar</code>	Desanexa o objeto encapsulado <code>VARIANT</code> deste _variant_t objeto.
<code>SetString</code>	Atribui uma cadeia de caracteres a este _variant_t objeto.

Operadores

NOME	DESCRIÇÃO
<code>Operador =</code>	Atribui um novo valor a um objeto de _variant_t existente.
<code>operador = =,! =</code>	Compare dois objetos _variant_t para igualdade ou desigualdade.
<code>Extratores</code>	Extrair dados do objeto encapsulado <code>VARIANT</code> .

FINAL específico da Microsoft

Requisitos

Cabeçalho: <comutil.h>

Lib: comsuppw.lib ou comsuppwd.lib (consulte [/Zc: Wchar_t \(Wchar_t é o tipo nativo\)](#) para obter mais informações)

Confira também

Funções de membro _variant_t

02/12/2019 • 2 minutes to read • [Edit Online](#)

Para obter informações sobre o **variant_t** funções de membro, consulte [classe variant_t](#).

Consulte também

[Classe variant_t](#)

_variant_t::variant_t

02/09/2020 • 8 minutes to read • [Edit Online](#)

Específico da Microsoft

Constrói um objeto `_variant_t`.

Sintaxe

```
_variant_t( ) throw( );

_variant_t(
    const VARIANT& varSrc
);

_variant_t(
    const VARIANT* pVarSrc
);

_variant_t(
    const _variant_t& var_t_src
);

_variant_t(
    VARIANT& varSrc,
    bool fCopy
);

_variant_t(
    short sSrc,
    VARTYPE vtSrc = VT_I2
);

_variant_t(
    long lSrc,
    VARTYPE vtSrc = VT_I4
);

_variant_t(
    float fltSrc
) throw( );

_variant_t(
    double dblSrc,
    VARTYPE vtSrc = VT_R8
);

_variant_t(
    const CY& cySrc
) throw( );

_variant_t(
    const _bstr_t& bstrSrc
);

_variant_t(
    const wchar_t *wstrSrc
);

_variant_t(
    const char* strSrc
);
```

```

);

_variant_t(
    IDispatch* pDispSrc,
    bool fAddRef = true
) throw( );

_variant_t(
    bool bSrc
) throw( );

_variant_t(
    IUnknown* pIUnknownSrc,
    bool fAddRef = true
) throw( );

_variant_t(
    const DECIMAL& decSrc
) throw( );

_variant_t(
    BYTE bSrc
) throw( );

variant_t(
    char cSrc
) throw();

_variant_t(
    unsigned short usSrc
) throw();

_variant_t(
    unsigned long ulSrc
) throw();

_variant_t(
    int iSrc
) throw();

_variant_t(
    unsigned int uiSrc
) throw();

_variant_t(
    __int64 i8Src
) throw();

_variant_t(
    unsigned __int64 ui8Src
) throw();

```

parâmetros

{1>varSrc<1}

Um objeto `VARIANT` a ser copiado no novo objeto `_variant_t`.

pVarSrc

Ponteiro para um `VARIANT` objeto a ser copiado para o novo `_variant_t` objeto.

var_t_Src

Um objeto `_variant_t` a ser copiado no novo objeto `_variant_t`.

fCopy

Se `false`, o `VARIANT` objeto fornecido será anexado ao novo `_variant_t` objeto sem fazer uma nova cópia `VariantCopy`.

ISrc, sSrc

Um valor inteiro a ser copiado no novo objeto `_variant_t`.

vtSrc

O `VARTYPE` para o novo `_variant_t` objeto.

fltSrc, dblSrc

Um valor numérico a ser copiado no novo objeto `_variant_t`.

cySrc

Um objeto `cy` a ser copiado no novo objeto `_variant_t`.

bstrSrc

Um objeto `_bstr_t` a ser copiado no novo objeto `_variant_t`.

strSrc, wstrSrc

Uma cadeia de caracteres a ser copiada no novo objeto `_variant_t`.

bSrc

Um `bool` valor a ser copiado para o novo `_variant_t` objeto.

pIUnknownSrc

Ponteiro de interface COM para um objeto VT_UNKNOWN a ser encapsulado no novo `_variant_t` objeto.

pDispSrc

Ponteiro de interface COM para um objeto VT_DISPATCH a ser encapsulado no novo `_variant_t` objeto.

decSrc

Um valor `DECIMAL` a ser copiado no novo objeto `_variant_t`.

bSrc

Um valor `BYTE` a ser copiado no novo objeto `_variant_t`.

cSrc

Um `char` valor a ser copiado para o novo `_variant_t` objeto.

usSrc

Um `unsigned short` valor a ser copiado para o novo `_variant_t` objeto.

ulSrc

Um `unsigned long` valor a ser copiado para o novo `_variant_t` objeto.

iSrc

Um `int` valor a ser copiado para o novo `_variant_t` objeto.

uiSrc

Um `unsigned int` valor a ser copiado para o novo `_variant_t` objeto.

i8Src

Um `_int64` valor a ser copiado para o novo `_variant_t` objeto.

ui8Src

Um valor de `_int64` não assinado a ser copiado para o novo `_variant_t` objeto.

Comentários

- `_variant_t ()` Constrói um objeto vazio `_variant_t`, `VT_EMPTY`.
- `_variant_t (variant& varSrc***)*` Constrói um `_variant_t` objeto a partir de uma cópia do `VARIANT`

objeto. O tipo de variante é mantido.

- `_variant_t (Variant * pVarSrc***)`* constrói um `_variant_t` objeto a partir de uma cópia do `VARIANT` objeto. O tipo de variante é mantido.
- `_variant_t (_variant_t& var_t_Src***)`* Constrói um `_variant_t` objeto de outro `_variant_t` objeto. O tipo de variante é mantido.
- `_variant_t (Variant& varSrc, bool fCopy)` constrói um `_variant_t` objeto de um objeto existente `VARIANT`. Se `fCopy` for `false`, o objeto `Variant` será anexado ao novo objeto sem fazer uma cópia.
- `*_variant_t (Short***sSrc, VARTYPE vtSrc = VT_I2)` constrói um `_variant_t` objeto do tipo `VT_I2` ou `VT_BOOL` de um `short` valor inteiro. Qualquer outro `VARTYPE` resulta em um erro de `E_INVALIDARG`.
- `_variant_t (Long lSrc, VARTYPE vtSrc = VT_I4)` constrói um `_variant_t` objeto do tipo `VT_I4`, `VT_BOOL` ou `VT_ERROR` de um `long` valor inteiro. Qualquer outro `VARTYPE` resulta em um erro de `E_INVALIDARG`.
- `_variant_t (float fltSrc)` constrói um `_variant_t` objeto do tipo `VT_R4` a partir de um `float` valor numérico.
- `_variant_t (Double dblSrc, VARTYPE vtSrc = VT_R8)` constrói um `_variant_t` objeto do tipo `VT_R8` ou `VT_DATE` de um `double` valor numérico. Qualquer outro `VARTYPE` resulta em um erro de `E_INVALIDARG`.
- `_variant_t (CY& cySrc)` constrói um `_variant_t` objeto do tipo `VT_CY` de um `CY` objeto.
- `_variant_t (_bstr_t& bstrSrc)` constrói um `_variant_t` objeto do tipo `VT_BSTR` de um `_bstr_t` objeto. Um novo `BSTR` é alocado.
- `_variant_t (wchar_t * wstrSrc)` constrói um `_variant_t` objeto do tipo `VT_BSTR` de uma cadeia de caracteres Unicode. Um novo `BSTR` é alocado.
- `_variant_t (Char * strSrc)` Constrói um `_variant_t` objeto do tipo `VT_BSTR` de uma cadeia de caracteres. Um novo `BSTR` é alocado.
- `_variant_t (bool bSrc)` constrói um `_variant_t` objeto do tipo `VT_BOOL` a partir de um `bool` valor.
- `_variant_t (IUnknown * pIUnknownSrc, bool fAddRef = true)` Constrói um `_variant_t` objeto do tipo `VT_UNKNOWN` de um ponteiro de interface com. Se `fAddRef` for `true`, `AddRef` será chamado no ponteiro de interface fornecido para corresponder à chamada para `Release` que ocorrerá quando o `_variant_t` objeto for destruído. Cabe a você chamar `Release` o ponteiro de interface fornecido. Se `fAddRef` for `false`, esse construtor assumirá a propriedade do ponteiro de interface fornecido; não chame `Release` no ponteiro de interface fornecido.
- `_variant_t (IDispatch * pDispSrc, bool fAddRef = true)` Constrói um `_variant_t` objeto do tipo `VT_DISPATCH` de um ponteiro de interface com. Se `fAddRef` for `true`, `AddRef` será chamado no ponteiro de interface fornecido para corresponder à chamada para `Release` que ocorrerá quando o `_variant_t` objeto for destruído. Cabe a você chamar `Release` o ponteiro de interface fornecido. Se `fAddRef` for `false`, esse construtor assumirá a propriedade do ponteiro de interface fornecido; não chame `Release` no ponteiro de interface fornecido.
- `_variant_t (decimal& decSrc)` constrói um `_variant_t` objeto do tipo `VT_DECIMAL` a partir de um `DECIMAL` valor.
- `_variant_t (byte bSrc)` constrói um `_variant_t` objeto do tipo `VT_UI1` a partir de um `BYTE` valor.

Confira também

[Classe_variant_t](#)

_variant_t::Attach

24/04/2020 • 2 minutes to read • [Edit Online](#)

Específico da Microsoft

Anexa um `VARIANT` objeto ao objeto `_variant_t`.

Sintaxe

```
void Attach(VARIANT& varSrc);
```

Parâmetros

`{1>varSrc<1}`

Um `VARIANT` objeto a ser anexado a este **objeto `_variant_t`**.

Comentários

Toma posse `VARIANT` do encapsulando-o. Esta função membro libera qualquer `VARIANT` encapsulamento existente, `VARIANT` depois copia `VARTYPE` o fornecido e define seu `VT_EMPTY` para garantir que seus recursos só possam ser liberados pelo **destruidor de `_variant_t`**.

Fim específico da Microsoft

Confira também

[Classe `_variant_t`](#)

_variant_t::Clear

24/04/2020 • 2 minutes to read • [Edit Online](#)

Específico da Microsoft

Limpa o objeto `VARIANT` encapsulado.

Sintaxe

```
void Clear( );
```

Comentários

Chama `VariantClear` o objeto `VARIANT` encapsulado.

Fim específico da Microsoft

Confira também

[Classe _variant_t](#)

_variant_t::ChangeType

24/04/2020 • 2 minutes to read • [Edit Online](#)

Específico da Microsoft

Altera o tipo `_variant_t` do objeto `VARTYPE` para o indicado .

Sintaxe

```
void ChangeType(
    VARTYPE vartype,
    const _variant_t* pSrc = NULL
);
```

Parâmetros

Vartype

O `VARTYPE` para `_variant_t` este objeto.

Psrc

Um ponteiro para o objeto `_variant_t` a ser convertido. Se esse valor for NULO, a conversão será feita no lugar.

Comentários

Esta função de `_variant_t` membro converte `VARTYPE` um objeto no indicado . Se o *pSrc* for NULL, a conversão `_variant_t` será feita no lugar, caso contrário este objeto será copiado do *pSrc* e convertido.

Fim específico da Microsoft

Confira também

[Classe _variant_t](#)

_variant_t::Detach

25/03/2020 • 2 minutes to read • [Edit Online](#)

Seção específica da Microsoft

Desanexa o objeto de `VARIANT` encapsulado deste objeto `_variant_t`.

Sintaxe

```
VARIANT Detach( );
```

Valor retornado

O `VARIANT` encapsulado.

Comentários

Extrai e retorna o `VARIANT` encapsulado e, em seguida, limpa esse objeto `_variant_t` sem destruí-lo. Essa função de membro remove a `VARIANT` do encapsulamento e define a `VARTYPE` desse objeto `_variant_t` como `VT_EMPTY`. Cabe a você liberar o `VARIANT` retornado chamando a função `VariantClear`.

Fim da seção específica da Microsoft

Confira também

[Classe _variant_t](#)

_variant_t::SetString

22/04/2020 • 2 minutes to read • [Edit Online](#)

Específico da Microsoft

Atribui uma cadeia de caracteres a este objeto `_variant_t`.

Sintaxe

```
void SetString(const char* pSrc);
```

Parâmetros

Psrc

Ponteiro para a cadeia de caracteres.

Comentários

Converte uma cadeia de caracteres ANSI em uma cadeia de caracteres Unicode `BSTR` e a atribui a este objeto `_variant_t`.

Fim específico da Microsoft

Confira também

[Classe _variant_t](#)

Operadores (_variant_t)

25/03/2020 • 2 minutes to read • [Edit Online](#)

Para obter informações sobre os operadores de `_variant_t`, consulte [_variant_t classe](#).

Confira também

[Classe _variant_t](#)

_variant_t::operator =

02/09/2020 • 2 minutes to read • [Edit Online](#)

Específico da Microsoft

Sintaxe

```
_variant_t& operator=(  
    const VARIANT& varSrc  
>);  
  
_variant_t& operator=(  
    const VARIANT* pVarSrc  
>);  
  
_variant_t& operator=(  
    const _variant_t& var_t_src  
>);  
  
_variant_t& operator=(  
    short sSrc  
>);  
  
_variant_t& operator=(  
    long lSrc  
>);  
  
_variant_t& operator=(  
    float fltSrc  
>);  
  
_variant_t& operator=(  
    double dblSrc  
>);  
  
_variant_t& operator=(  
    const CY& cySrc  
>);  
  
_variant_t& operator=(  
    const _bstr_t& bstrSrc  
>);  
  
_variant_t& operator=(  
    const wchar_t* wstrSrc  
>);  
  
_variant_t& operator=(  
    const char* strSrc  
>);  
  
_variant_t& operator=(  
    IDispatch* pDispSrc  
>);  
  
_variant_t& operator=(  
    bool bSrc  
>);  
  
_variant_t& operator=(  
    IUnknown* pSrc  
>);
```

```

);

_variant_t& operator=(
    const DECIMAL& decSrc
);

_variant_t& operator=(
    BYTE bSrc
);

_variant_t& operator=(
    char cSrc
);

_variant_t& operator=(
    unsigned short usSrc
);

_variant_t& operator=(
    unsigned long ulSrc
);

_variant_t& operator=(
    int iSrc
);

_variant_t& operator=(
    unsigned int uiSrc
);

_variant_t& operator=(
    __int64 i8Src
);

_variant_t& operator=(
    unsigned __int64 ui8Src
);

```

Comentários

O operador atribui um novo valor ao objeto `_variant_t`:

- **Operator = (varSrc)** Atribui um existente `VARIANT` a um `_variant_t` objeto.
- **Operator = (pVarSrc)** Atribui um existente `VARIANT` a um `_variant_t` objeto.
- **Operator = (var_t_Src)** Atribui um `_variant_t` objeto existente a um `_variant_t` objeto.
- **Operator = (sSrc)** Atribui um `short` valor inteiro a um `_variant_t` objeto.
- **Operator = (lSrc)** atribui um `long` valor inteiro a um `_variant_t` objeto.
- **Operator = (fltSrc)** Atribui um `float` valor numérico a um `_variant_t` objeto.
- **Operator = (dblSrc)** Atribui um `double` valor numérico a um `_variant_t` objeto.
- **Operator = (cySrc)** Atribui um `cy` objeto a um `_variant_t` objeto.
- **Operator = (bstrSrc)** Atribui um `BSTR` objeto a um `_variant_t` objeto.
- **Operator = (wstrSrc)** Atribui uma cadeia de caracteres Unicode a um `_variant_t` objeto.
- **Operator = (strSrc)** atribui uma cadeia de caracteres multibyte a um `_variant_t` objeto.
- **Operator = (bSrc)** atribui um `bool` valor a um `_variant_t` objeto.

- **Operator = (*pDispSrc*)** Atribui um `VT_DISPATCH` objeto a um `_variant_t` objeto.
- **Operator = (*pIUnknownSrc*)** Atribui um `VT_UNKNOWN` objeto a um `_variant_t` objeto.
- **Operator = (*decSrc*)** Atribui um `DECIMAL` valor a um `_variant_t` objeto.
- **Operator = (*bSrc*)** atribui um `BYTE` valor a um `_variant_t` objeto.

FINAL específico da Microsoft

Confira também

[Classe _variant_t](#)

Operadores relacionais `_variant_t`

02/09/2020 • 2 minutes to read • [Edit Online](#)

Específico da Microsoft

Compare dois objetos `_variant_t` em termos de igualdade ou desigualdade.

Sintaxe

```
bool operator==(  
    const VARIANT& varSrc) const;  
bool operator==(  
    const VARIANT* pSrc) const;  
bool operator!=(  
    const VARIANT& varSrc) const;  
bool operator!=(  
    const VARIANT* pSrc) const;
```

parâmetros

`{1>varSrc<1}`

Um `VARIANT` a ser comparado com o `_variant_t` objeto.

`pSrc`

Ponteiro para o `VARIANT` a ser comparado com o `_variant_t` objeto.

Valor retornado

Retorna `true` se a comparação se aplica, `false` caso contrário.

Comentários

Compara um `_variant_t` objeto com um `VARIANT`, testando a igualdade ou desigualdade.

FINAL específico da Microsoft

Confira também

[Classe `_variant_t`](#)

Extratores _variant_t

02/09/2020 • 2 minutes to read • [Edit Online](#)

Específico da Microsoft

Extrair dados do objeto encapsulado `VARIANT`.

Sintaxe

```
operator short( ) const;
operator long( ) const;
operator float( ) const;
operator double( ) const;
operator CY( ) const;
operator _bstr_t( ) const;
operator IDispatch*( ) const;
operator bool( ) const;
operator IUnknown*( ) const;
operator DECIMAL( ) const;
operator BYTE( ) const;
operator VARIANT() const throw();
operator char() const;
operator unsigned short() const;
operator unsigned long() const;
operator int() const;
operator unsigned int() const;
operator __int64() const;
operator unsigned __int64() const;
```

Comentários

Extrai dados brutos de um encapsulado `VARIANT`. Se o `VARIANT` não for o tipo correto, `VariantChangeType` será usado para tentar uma conversão e um erro será gerado após a falha:

- **operador Short ()** Extrai um `short` valor inteiro.
- **operador Long ()** Extrai um `long` valor inteiro.
- **operador float ()** Extrai um `float` valor numérico.
- **operador Double ()** Extrai um `double` valor inteiro.
- **operador CY ()** Extrai um `CY` objeto.
- **operador bool ()** Extrai um `bool` valor.
- **Operador Decimal ()** Extrai um `DECIMAL` valor.
- **byte do operador ()** Extrai um `BYTE` valor.
- **_bstr_t do operador ()** Extrai uma cadeia de caracteres, que é encapsulada em um `_bstr_t` objeto.
- **o operador IDispatch * ()** extrai um ponteiro de dispinterface de um encapsulado `VARIANT`. `AddRef` é chamado no ponteiro resultante, portanto, cabe a você chamar `Release` para liberá-lo.
- **o operador IUnknown * ()** extrai um ponteiro de interface com de um encapsulado `VARIANT`. `AddRef` é

chamado no ponteiro resultante, portanto, cabe a você chamar `Release` para liberá-lo.

FINAL específico da Microsoft

Confira também

[Classe _variant_t](#)

Extensões da Microsoft

02/09/2020 • 2 minutes to read • [Edit Online](#)

```
asm-statement :  
* __asm *** assembly-instruction ; aceitar  
* __asm { *** assembly-instruction-list } ; aceitar  
  
assembly-instruction-list :  
assembly-instruction *** ; * aceitar  
assembly-instruction *** ; * assembly-instruction-list ; aceitar  
  
ms-modifier-list :  
ms-modifier ** ms-modifier-list aceitar  
  
ms-modifier :  
__cdecl  
__fastcall  
__stdcall  
__syscall (reservado para implementações futuras)  
__oldcall (reservado para implementações futuras)  
__unaligned (reservado para implementações futuras)  
based-modifier  
  
based-modifier :  
__based ( based-type )  
  
based-type :  
name
```

Comportamento não padrão

02/09/2020 • 3 minutes to read • [Edit Online](#)

As seções a seguir listam alguns dos locais em que a implementação do C++ da Microsoft não está em conformidade com o padrão C++. Os números de seção fornecidos abaixo se referem aos números da seção no padrão C++ 11 (ISO/IEC 14882:2011(E)).

A lista de limites de compilador que diferem daqueles definidos no padrão C++ é fornecida nos [limites do compilador](#).

Tipos de retorno covariantes

As classes base virtuais não têm suporte como tipo de retorno covariante quando a função virtual tem um número variável de argumentos. Isso não está em conformidade com a seção 10.3, parágrafo 7 da especificação ISO do C++. O exemplo a seguir não compila, dando erro do compilador [C2688](#)

```
// CovariantReturn.cpp
class A
{
    virtual A* f(int c, ...); // remove ...
};

class B : virtual A
{
    B* f(int c, ...); // C2688 remove ...
};
```

Associação de nomes não dependentes nos modelos

Atualmente, o compilador do Microsoft C++ não dá suporte à associação de nomes não dependentes ao analisar inicialmente um modelo. Isso não está em conformidade com a seção 14.6.3 da especificação ISO do C++. Isso pode fazer com que sobrecargas declaradas após o modelo ser instanciado sejam vistas.

```
#include <iostream>
using namespace std;

namespace N {
    void f(int) { cout << "f(int)" << endl;}
}

template <class T> void g(T) {
    N::f('a'); // calls f(char), should call f(int)
}

namespace N {
    void f(char) { cout << "f(char)" << endl;}
}

int main() {
    g('c');
}
// Output: f(char)
```

Especificadores de exceção de função

Os especificadores da exceção de função diferentes de `throw()` são analisados, mas não usados. Isso não está em conformidade com a seção 15.4 da especificação ISO do C++. Por exemplo:

```
void f() throw(int); // parsed but not used
void g() throw();    // parsed and used
```

Para obter mais informações sobre especificações de exceção, consulte [especificações de exceção](#).

char_traits::eof()

O padrão C++ declara que `char_traits::EOF` não deve corresponder a um `char_type` valor válido. O compilador do Microsoft C++ impõe essa restrição para o tipo `char`, mas não para o tipo `wchar_t`. Isso não está em conformidade com o requisito da Tabela 62, na seção 12.1.1 da especificação ISO do C++. O exemplo abaixo demonstra isso.

```
#include <iostream>

int main()
{
    using namespace std;

    char_traits<char>::int_type int2 = char_traits<char>::eof();
    cout << "The eof marker for char_traits<char> is: " << int2 << endl;

    char_traits<wchar_t>::int_type int3 = char_traits<wchar_t>::eof();
    cout << "The eof marker for char_traits<wchar_t> is: " << int3 << endl;
}
```

Local de armazenamento de objetos

O padrão C++ (seção 1.8, parágrafo 6) exige que objetos completos do C++ tenham locais exclusivos de armazenamento. No entanto, com o Microsoft C++, há casos em que tipos sem membros de dados compartilharão um local de armazenamento com outros tipos para o tempo de vida do objeto.

Limites do compilador

02/09/2020 • 2 minutes to read • [Edit Online](#)

O padrão do C++ recomenda limites para várias construções de linguagem. Veja a seguir uma lista de casos em que o compilador do Microsoft C++ não implementa os limites recomendados. O primeiro número é o limite estabelecido no padrão ISO C++ 11 (INCITS/ISO/IEC 14882-2011 [2012], anexo B) e o segundo número é o limite implementado pelo compilador do Microsoft C++:

- Aninhando níveis de instruções compostas, estruturas de controle de iteração e estruturas de controle de seleção-C++ Standard: 256, compilador do Microsoft C++: depende da combinação de instruções que estão aninhadas, mas geralmente entre 100 e 110.
- Parâmetros em uma definição de macro-C++ Standard: 256, compilador do Microsoft C++: 127.
- Argumentos em uma única macro invoca-C++ Standard: 256, compilador do Microsoft C++ 127.
- Caracteres em um literal de cadeia de caracteres ou literal de cadeia de caracteres largo (após concatenação)-C++ Standard: 65536, compilador do Microsoft C++: 65535 caracteres de byte único, incluindo o terminador nulo e 32767 caracteres de byte duplo, incluindo o terminador nulo.
- Níveis de definições de classe aninhada, estrutura ou União em um `struct-declaration-list` padrão de C++ único: 256, compilador do Microsoft C++: 16.
- Inicializadores de membro em uma definição de construtor-C++ Standard: 6144, compilador do Microsoft C++: pelo menos 6144.
- Qualificações de escopo de um identificador-C++ Standard: 256, compilador do Microsoft C++: 127.
- Especificações aninhadas `extern` -C++ Standard: 1024, compilador do Microsoft C++: 9 (sem contar a `extern` especificação implícita no escopo global, ou 10, se você contar a `extern` especificação implícita no escopo global...)
- Argumentos de modelo em uma declaração de modelo-C++ Standard: 1024, compilador do Microsoft C++: 2046.

Confira também

[Comportamento não padrão](#)

Referência de pré-processador C/C++

02/12/2019 • 2 minutes to read • [Edit Online](#)

A *referência C/C++ /pré-processador* explica o pré-processador conforme ele é implementado no Microsoft C/C++. O pré-processador executa operações preliminares em arquivos do C e C++ antes de serem passados para o compilador. É possível usar o pré-processador para compilar o código condicionalmente, inserir arquivos, especificar mensagens de erro de tempo de compilação e aplicar regras de máquina específicas para seções de código.

No Visual Studio 2019, a opção de compilador [/experimental:pré-processador](#) permite uma nova implementação do pré-processador. A nova implementação ainda está em andamento e, portanto, é considerada experimental. A intenção é, eventualmente, ser compatível com C99, C11 e C++ 20. Para obter mais informações, consulte [visão geral do pré-processador do MSVC experimental](#).

Nesta seção

\ de [pré-processador](#)

\ de [diretivas de pré-processador](#)

[Operadores de pré-processador](#)

Discute os quatro operadores específicos de pré-processadores usados no contexto da política `#define`.

[Macros Predefinidas](#)

Discute macros predefinidas conforme especificado por ANSI e Microsoft C++.

[Pragmas](#)

Discute pragmas, que proporcionam uma maneira para que cada compilador ofereça recursos específicos de máquinas e sistemas operacionais enquanto mantém a compatibilidade geral com as linguagens C e C++.

Seções relacionadas

[referência de linguagem\ C++](#)

\ de [referência de linguagem C](#)

\ de [referência C/C++ Build](#)

[Projetos do Visual Studio C++ -](#)

Descreve a interface do usuário no Visual Studio que permite especificar os diretórios que o sistema do projeto procurará para localizar arquivos para o seu projeto do C++.

Referência da Biblioteca Padrão C++

02/12/2019 • 2 minutes to read • [Edit Online](#)

É possível chamar um programa do C++ em um grande número de funções por meio dessa implementação em conformidade da Biblioteca Padrão C++. Essas funções realizam serviços essenciais como entrada e saída, além de oferecer implementações eficientes das operações mais usadas.

Para obter mais informações sobre as bibliotecas em tempo de execução Visual C++, consulte [Recursos da biblioteca CRT](#).

Nesta seção

[Visão geral da biblioteca padrão C++](#)

Fornece uma visão geral da implementação da Microsoft da Biblioteca Padrão C++.

[Programação de iostream](#)

Fornece uma visão geral da programação de iostream.

[Referência de Arquivos de Cabeçalho](#)

Fornece links para tópicos de referência que abordam os arquivos de cabeçalho da Biblioteca Padrão C++, com exemplos de código.